1

THE GRAPHICAL REPRESENTATION OF ALGORITHMIC PROCESSES VOLUME 1

THESIS

Keith Carson Fife
Captain, USAF

AFIT/GCE/ENG/89D-2

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

1

AFIT/GCE/ENG/89D-2

THE GRAPHICAL REPRESENTATION OF ALGORITHMIC PROCESSES VOLUME 1

THESIS

Keith Carson Fife
Captain, USAF

AFIT/GCE/ENG/89D-2

AFIT/GCE/ENG/89D-2

THE GRAPHICAL REPRESENTATION OF ALGORITHMIC PROCESSES VOLUME 1

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Engineering

Keith Carson Fife, B.S.
Captain, USAF

December, 1989

Preface

The goal of my research was to develop an impressive algorithm animation system using the techniques of modern software engineering. The AFIT Algorithm Animation Research Facility, or AAARF, is the result of that research. As for its impressiveness, few events in academia draw a crowd of excited researchers as quickly as the thrill of an AAARF algorithm race. With respect to the development, the classic life-cycle paradigm for software engineering was used; IDEF$_0$ diagrams, enumerated requirements specifications, object-oriented design concepts, and structure charts were used to document the effort. Over 10,000 lines of code were written, roughly half of which are comments.

I especially want to thank Marc Brown for his pioneering work in algorithm animation. His book, *Algorithm Animation*, provided ideas for many of AAARF's features. AAARF differs from Brown's BALSA system in that it uses multiple processes to animate algorithms and supports remotely-hosted algorithm processes.

I want to thank my advisor, Dr Gary Lamont, for allowing me almost complete freedom over the design and implementation of AAARF, but providing just the right nudges to keep me from using that freedom to "hang" myself. I must also thank Major Phil Amburn and Dr Thomas Hartrum for their excellent instruction in Computer Graphics and Software Engineering. Without the use of Major Amburn's specially-configured Sun4 workstation, AAARF would not be what it is today.

I want to acknowledge my gratitude to nearly everyone I know – my family in Florida, my friends in California, and the Friday night gang here in Dayton. Without their support and comradery, I couldn't get through life much less a thesis. Finally, if anyone deserves gratitude, it's my son, Bryan, who endured a year with a father preoccupied with "something called AAARF." Thanks, Bryan.

Keith Carson Fife

Table of Contents

List of Figures

AFIT/GCE/ENG/89D-2

Abstract

Traditionally, software engineers have described algorithms and data structures using graphical representations such as flow charts, structure charts, and block diagrams. These representations can give a static general overview of an algorithm, but they fail to fully illustrate an algorithm's dynamic behavior. Researchers have begun to develop systems to visualize, or animate, algorithms in execution. The form of the visualization depends on the algorithm being examined, the data structures being used, and the ingenuity of the programmer implementing the animation.

This study chronicles the development of the AFIT Algorithm Animation Research Facility (AAARF). The goal of the study is (1) to develop a method for developing algorithm animations and (2) to develop an environment in which the animations can be displayed. The study emphasizes the application of modern software engineering techniques. The development follows a classic life-cycle software engineering paradigm: requirements, design, implementation, and testing. An object-oriented approach is used for the preliminary design. The system is implemented with the C programming language on a Sun workstation and uses the SunView window-based environment.

A framework is proposed for developing algorithm animations that minimizes the development time for client-programmers. The framework also ensures that end-users are presented a consistent method for interacting with the animations. The algorithm animation environment provides end-users with a variety of control and viewing options: multiple algorithms, multiple views, simultaneous control of algorithms, and animation environment control. Three levels of execution are used to provide multiple algorithm animations and multiple views of algorithms. The animation manager and view generators must reside on the Sun workstation, but the algorithms can reside on any network-connected host.

THE GRAPHICAL REPRESENTATION OF ALGORITHMIC PROCESSES VOLUME 1

I. Introduction

1.1 Background

An algorithm animation system is a "software microscope" for analyzing the dynamic behavior of algorithms and their related data structures. By dynamically displaying a graphical representation of an executi .g algorithm's current state, animation can reveal properties of the algorithm that are not easily expressed in pseudocode, flow charts, or data-flow diagrams. In fact, unknown, perhaps unwanted, properties can be revealed [6:5], helping software engineers "see" ways to realize more efficient and effective algorithms. As an educational tool, algorithm animation can help students understand new and difficult algorithms by providing a means for visualizing and interacting with algorithms as they execute. As a production tool, animation provides a means for visually analyzing the impact of particular algorithms on specific problems and data sets, helping develop new and better solutions for increasingly difficult programming challenges.

1.2 Purpose

The purpose of this study is (1) to develop a methodology for creating algorithm animations and (2) to develop an environment for controlling, displaying, and interacting with the animations. This dual purpose reflects the needs of two types of users: *"client-programmers"* and *"end-users"* [6:6].

Client-programmers are concerned with implementing the algorithm animations with which end-users interact. With respect to client-programmers, the goal of this study is to create a program development environment which provides a consistent interface to the algorithm animation system and supports reusable software modules. The programmer should not have to reimplement modules common to several algorithms, such as window management, user-interface, and display functions [6:7]. Further, the development environment should provide a set of primitives that are natural to animation; for example, smooth continuous movement along a path [23].

End-users view and interact with the animations at a computer workstation. With respect to end-users, the goal of this investigation is to provide an algorithm animation run-time environment which provides a consistent method for interacting with the animations. After animating one algorithm, end-users should be able to animate any algorithm, regardless of the type of algorithm or the client-programmer of the animation [6:7].

1.3 Problem

In general, the problem is to *develop a method for animating any algorithm.* Three general approaches for solving the problem are proposed:

- **Client-Programmer Based.** Develop a library of support procedures which the client-programmer uses along with a set of algorithm-specific procedures to produce an executable algorithm animation. This approach creates a separate executable program for every algorithm animation. No provisions are made for controlling or viewing multiple algorithms, and the user has no way of knowing what animations are available. Though client-programmers can easily develop algorithm animations, the end-user interface is poor.

- **End-User Based.** Develop an animation system which animates any algorithm based on some set of parameters which describe the algorithm. The parameters are developed by the client-programmer. This approach requires only one executable program with a separate parameter list for every algorithm to be animated. The system manages the parameter lists so that end-users can easily select from a collection of available algorithms. The system provides a mechanism for controlling and viewing multiple animations. But since the animations are generated based on a set of parameters which describe the algorithm, the animations are necessarily simple and potentially ineffective. Though complex animations might be possible, the parameterized description would take a herculean effort to produce. This approach provides a better end-user interface, but presents a problem for the client-programmer.

- **Dual-Interface.** The third approach incorporates aspects of both the end-user based and client-programmer based approaches. Develop a system through which a user can select, execute, and control individual animations, each of which is a separate executable procedure. The client-programmer develops the executable procedures with the help of a library of algorithm animation support functions. This approach considers both the end-user and client-programmer interfaces.

This investigation pursues the dual-interface approach. The goal is to develop an algorithm animation environment which presents an easy-to-use, functional interface to end-users, and provides an effective means for managing algorithm animations within the animation environment for client-programmers.

1.4 Scope

Although many algorithms can be animated to test and demonstrate the system, the goal of this project is not algorithm research. Rather, the goal is to develop a system on which algorithm research can be conducted and to develop a methodology for creating algorithm animations. The emphasis is not on product completeness, but on the proper application of modern software engineering principles toward the development of that product.

1.5 Assumptions

1. Algorithm animation displays cannot be created automatically. Algorithm animation can do more than just monitor the value of a program variable; it can present abstract graphical representations of an algorithm's fundamental operations. These fundamental operations cannot be deduced for an arbitrary algorithm; they must be defined by a programmer familiar with the operation of the algorithm [6:18]. Likewise the selection and development of a graphical representation cannot be performed automatically. Some specialized data structures and algorithms require unique representations to make aspects of their execution understandable [6:19].

2. Any algorithm can be animated. All algorithms manipulate some non-empty set of data structures, and at some level of abstraction, every data structure can be represented graphically - monitoring and displaying the data structure's value being the lowest. However, arbitrary algorithm animations are not necessarily effective. Many factors impact the effectiveness of an animation: meaningful input data, proper identification of interesting events, meaningful views of the algorithm state, appropriate level of end-user interaction.

3. An algorithm animation system can be implemented on a Sun3^{TM}, Sun4^{TM}, or compatible workstation. Since it is the workstation of choice at AFIT, a Sun-based algorithm animation system provides the widest possible local user-base. Beyond that, Sun provides an extensive window-based environment for the development of user interfaces in SunView^{TM} [28]. Sun also provides a variety of graphics packages including SunCore^{TM}, SunCGI^{TM}, and Sun Pixrect [26].

4. An algorithm animation system can be developed using the C programming language. Since SunView is written in C, and the Sun operating system, SunOS^{TM} [24], provides an extensive C library, C is an obvious choice as the development language. As a programming language, C is sufficiently low-level to provide the execution speed required by a graphics intensive program, and sufficiently high-level to support modern software engineering principles[4:31]. A high-level language such as Ada^{TM} or Pascal might not provide the necessary speed.

1.6 Overview

This investigation approaches the problem of developing an algorithm animation system as a sequence of logical stages. The stages are not discrete, and the sequence is not necessarily chronological. Many of the stages overlap, and there is considerable feedback between stages.

A review of current literature and software relating to the graphical representation of algorithms and data structures is conducted with the goal of developing a knowledge-base from which the requirements analysis and preliminary design can be developed. The results of the literature search are presented in Section 2.3.

In the rapid prototype stage, several stand-alone sort animations are implemented to help identify the requirements of a general purpose animation system. The goal of this stage is to enhance the algorithm animation knowledge-base in preparation for the requirements analysis and preliminary design stages, and to become familiar with SunOS and SunView in preparation for the detailed design and implementation stages.

Based on the results of the literature review and rapid prototype, the requirements for an algorithm animation system are defined and documented. The goal of the requirements analysis is to determine:

- The requirements of the user interface,
- The requirements of the programmer interface,
- The functions and parameters required to control the algorithm animation environment, and
- The functions and parameters required to control the algorithm animations.

The requirements analysis discussion along with the enumerated requirements specification and $IDEF_0$ diagrams are presented in Section 2.4.

Based on the requirements analysis, a design is developed for a general-purpose algorithm animation system, the AFIT Algorithm Animation Research Facility (AAARF). An object-oriented approach is used for the preliminary AAARF design. Chapter III discusses the design stage.

Testing is conducted on several levels throughout the design and implementation. Design testing is discussed in Section 3.2, unit and integration testing in Section 4.4, and validation testing in Section 5.3.

With the rapid prototype as a starting point and the preliminary design as a development guide, AAARF is implemented on a Sun4 workstation. Structure charts are used to develop the hierarchical structure of the program design. The detailed design and implementation is presented in Chapter IV and detailed in the *AAARF Programmer's Guide* in Appendix D. The detailed design structure charts are presented in Appendix A.

Documentation is developed throughout all stages of the investigation. In Appendix C, the *AAARF User's Manual* describes how to use the algorithm animation system. Appendix D is the *AAARF Programmer's Guide*, it describes the algorithm animation system and presents a procedure for creating new algorithm animations. Appendix E is the AAARF source code, fully documented in accordance with AFIT System Development Documentation Guidelines and Standards [14]. Appendix E is included as a separate volume.

1.7 Summary

This chapter described the problem of *developing a method to animate algorithms*. The proposed approach for solving the problem considers the needs of two types of algorithm animation system users: end-users and client-programmers. A sequence of development stages was described for implementing the proposed approach. The remaining chapters present the details of the approach and the results of the effort.

II. *Requirements Analysis and Specification*

2.1 Introduction

This chapter presents the requirements analysis performed on the problem of developing an interactive algorithm animation system. Based on this analysis, the functional requirements for an interactive algorithm animation system are presented. Section 2.2 presents information intended to familiarize readers with the terms and concepts associated with algorithm animation. Previous and current research that has influenced this study is reviewed in Section 2.3. Section 2.4 presents the requirements analysis discussion. The enumerated requirements specification is included at the end of the chapter.

2.2 Background

A clear understanding of system definitions is a requirement for the analysis of any system. This section defines the concepts and terms associated with algorithms and algorithm animation. First, background information concerning algorithms and data structures is presented. Next, terms and concepts associated with algorithm animation are explained. Information from this section is used throughout the study and is critical to an exact understanding of this investigation.

Data Structures. Data structures along with algorithms, or control structures, are the basic building blocks of all software programs. Data structures represent the objects that programs manipulate; lists, queues, and stacks are commonly-used data structures. Horowitz and Sahni provide the following definitions regarding data structures:

> The *specification of a data structure* is a set of classes $\mathcal{D}$, a designated class d, a set of functions $\mathcal{F}$, and a set of axioms $\mathcal{A}$. The set of axioms describes the semantics of the set of functions, but imply nothing concerning their implementation. The triple $(\mathcal{D}, \mathcal{F}, \mathcal{A})$ denotes the data structure d, and is referred to as an ***abstract data type***.
>
> The *implementation of a data structure*, d, is a mapping from d to a set of other data structures, e. This mapping specifies how every object of d is to be represented by the objects of e. Further, it requires that every function of d be implemented using functions of the implementing data structures, e. [16:7]

An array is among the simplest examples of a data structure. Informally, an array is a set of index-value pairs. Figure 2.1 provides a more functional definition (specification) of an array. The set of classes, $\mathcal{D}$, is {*array, index, value*}. The designated class, d, is *array*. CREATE, RETRIEVE, and STORE establish the set of functions, $\mathcal{F}$, which are the fundamental operations for manipulating the *array* data structure. Finally, the **for all** section lists the set of axioms, $\mathcal{A}$, which define the operation of the functions and provide a means for proving the correctness of programs.

```
structure ARRAY(value, index)
    declare  CREATE() ⟶ array
             RETRIEVE(array, index) ⟶ value
             STORE(array, index, value) ⟶ array;
    for all $A \in$ array and $i, j \in$ index and $x \in$ value let
        RETRIEVE(CREATE,$i$) ::= error
        RETRIEVE(STORE($A, i, x$),$j$) ::=
             if EQUAL($i, j$) then $x$ else RETRIEVE($A, j$)
    end
end ARRAY
```

Figure 2.1. Specification of the ARRAY Data Structure [16:40]

The abstract data type (ADT) is introduced here as a prelude to a discussion of algorithm animation. In Chapter III, the subject of ADTs is revisited from an object-oriented design perspective. In object-oriented design, ADTs and their functions are referred to as *objects* and *operations*.

Control Structures. Control structures, or algorithms, are the heart of all software programs. Data structures represent the objects that programs manipulate, and algorithms represent the methods for manipulating those objects [3:2].

> An *algorithm*, or *control structure*, is a finite set of instructions which, if followed, accomplish a particular task. In addition, every algorithm must satisfy the following criteria:
>
> 1. *input:* there are zero or more quantities which are externally supplied;
> 2. *output:* at least one quantity is produced;
> 3. *definiteness:* each instruction must be clear and unambiguous;
> 4. *finiteness:* if the instructions of an algorithm are traced, then for all cases, the algorithm terminates after a finite number of steps;
> 5. *effectiveness:* every instruction must be sufficiently basic that it can, in principle, be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible. [16:2]

Figure 2.2 shows an algorithm for sorting an array of integers from smallest to largest. The input to the algorithm is the unsorted array. The output is the sorted array. For this simple algorithm, it is obvious that the criteria of definiteness, finiteness, and effectiveness are satisfied; for more complicated algorithms a formal proof may be required [10:7].

```
procedure BUBBLE_SORT($array[1..n]$ : $integer$);
    declare
        $i, j, temp$ : $integer$;
begin                                                - initArray
    for $i = 1$ to $n - 1$
        for $j = n$ to $i + 1$
            if ($array[j] < array[j-1]$)             - compare
                $temp = array[j]$;                   - exchange
                $array[j] = array[j-1]$;             -
                $array[j-1] = temp$;                 -
            end if
        end
    end                                              - inPlace
end BUBBLE_SORT                                      - finished
```

Figure 2.2. BUBBLE_SORT Algorithm

Algorithm Specifications. Animating the execution of an algorithm involves the development of meaningful graphical representations of the algorithm's fundamental operations. To this end, the idea of an algorithm specification is introduced. The algorithm specification incorporates portions of the data structure specification and the definition of an algorithm.

The *specification of an algorithm* can be given by the quad-tuple $\{D, I, S, E\}$ where D is a set of data structures, $I \subset D$ is a set of primary data structures, S is a sequence of instructions, and E is a set of *algorithm events.*

The set of data structures, D, refers to the set of objects manipulated by the algorithm, with I corresponding to those members of the set that are declared externally and imported into the algorithm. For the algorithm of Figure 2.2, the set of data structures consists of the (integer) array and (scalar) integer data types. The array is declared externally and is the only data structure (member of D) of importance outside the scope of the algorithm.

The set of algorithm events, or *interesting events*, E, consists of abstractions of the algorithm's fundamental operations. They can be input, output, or state transition events. Identifying algorithm events is similar to encapsulating an algorithm's operations into procedure calls such that there is a one-to-one mapping of events to procedures. In Figure 2.2, the algorithm events are identified as comments along the right side of the algorithm. **initArray** is an input event, **compare**, **exchange**, and **finished** are state transition events, and **inPlace** is an output event.

The sequence of instructions, S, invokes the algorithm events to accomplish the algorithm's task. The task is to transform D from some initial state, q_i, to some final state, q_f. S provides the transformation by invoking algorithm events.

$$q_i \xrightarrow{S} q_f$$

From an algorithm animation perspective, the algorithm events and intermediate states are more interesting than the final state alone. In this case, S transforms D from its initial state, q_i, through n intermediate states, to a final state, q_f, by invoking m algorithm events.

$$q_i \xrightarrow{S} q_f \times (E^m \times D^n)$$

Animating the algorithm consists of developing meaningful graphical representations of the algorithm events and intermediate states, $E^m \times D^n$.

Many systems have been developed that produce graphical representations of data structures [19, 5], but such systems fail to reveal *how* the data is being processed, and can present a confusing or misleading representation of the algorithm's function. For example, in an insertion sort, the algorithm searches a sorted array for an insertion point - no changes are made to the data structure until the insertion point is found. Only the insertion is displayed; *how* the insertion point was found is not. Algorithm animation can do more than simply monitor the value of a data structure; it can monitor an algorithm's fundamental operations and present a representation of the algorithm's state ($E \times D$), not just the state of the data structure D.

Algorithm Animation. In the few algorithm animation systems that have been developed [6, 1, 2, 17, 23], no definitive set of terms has emerged as a standard, neither has there been a standard algorithm animation system model. This section presents the algorithm animation model used in this investigation. The terms introduced here are used consistently throughout this thesis, but do not necessarily coincide with the terms used in other algorithm animation systems.

Algorithm Animation Components. This investigation assumes that an algorithm animation consists of three components: an input generator, an algorithm, and one or more views [6:8]. Figure 2.3 illustrates the utility of this decomposition; the individual components are independent and can be modularly replaced. This provides the flexibility required to support reusable components. Other decompositions have been tried [17, 19], but none are as flexible and intuitive as the input-algorithm-view paradigm.

The components operate much like UNIX pipes: the output of one component is the input to the next. The input generator provides data to the algorithm; the data may be generated randomly, read from a file, or entered by the user. The algorithm generates a stream of interesting events which drive the animation views. The views generate the animated displays of the algorithm's execution.

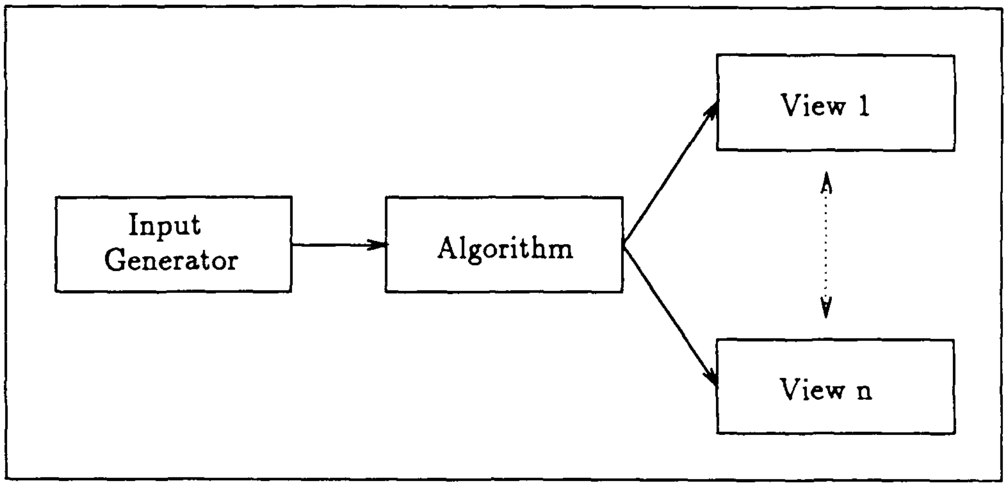


Figure 2.3. Algorithm Animation Components

Algorithm Classes. Algorithms which operate on identical data structures, and perform identical functions are from the same algorithm *class*. Using the D, I, S, E algorithm model, algorithm A and algorithm B are from the same algorithm class if and only if

$$I_A = I_B$$

and

$$q_i(I_A) = q_i(I_B) \longrightarrow q_f(I_A) = q_f(I_B)$$

The *sort* class includes ***quick sort***, ***heap sort***, ***bubble sort***, and other sort algorithms. Algorithms from the same class generally share input generators and views, although certain views and input generators may be ineffective with particular algorithms. For instance, a *tree* view is very effective with heap sort, but is of little use for any other sort.

Parameterized Control. User-selectable parameters are associated with each component. *"Algorithm parameters"* [6:8] affect some aspect of how the algorithm executes. For example, with a quick sort algorithm, what partitioning strategy should be used? As the partitions get smaller, at what point should another type of sort be used? *"Input Parameters"* [6:8] affect the input generator – what seed is used to generate a set of random numbers? How "sorted" is a set of unsorted numbers? What is the general form of a series of numbers? *"View parameters"* [6:8] affect how the animation is displayed in the view window. For example, what shape is associated with the nodes in a graph? How should an arbitrary graph be positioned? Figure 2.4 shows the algorithm component structure with parameterized control.

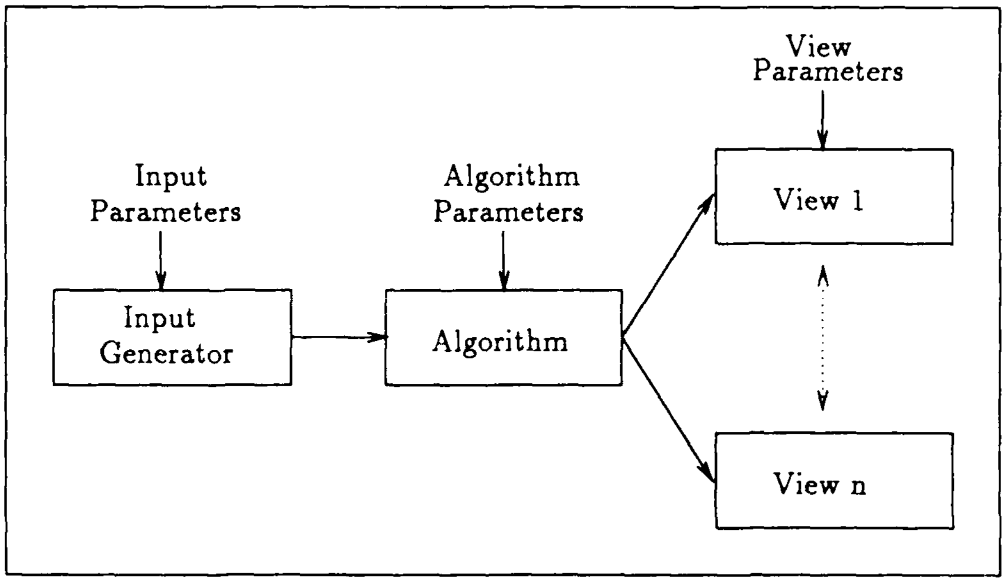


Figure 2.4. Parameterized Control of Algorithm Animation Components

Algorithm Windows. The portion of the workstation designated for the display of an algorithm animation is called an *algorithm window*. Within each algorithm window, multiple views of the algorithm may be displayed. The area of the algorithm window on which a view is displayed is called a *view window*.

Algorithm Animation System Model. This section presents an abstract model for an algorithm animation system. The model is based on the needs of both end-users and client-programmers, and incorporates the concepts of algorithm classes and components. The model provides a concise description of an algorithm animation system and represents a starting point for the requirements analysis and design of an algorithm animation system. The elements of an *algorithm animation system* are depicted in Figure 2.5 and include the following:

Algorithm Animation Environment Manager. The environment manager is a process with which end-users interact to select, control, and view algorithm animations. The environment manager supports multiple algorithm animations and provides a means for controlling their execution.

Algorithm Animation Component Library. The component library is a collection of algorithm animation components arranged by algorithm class. The environment manager invokes procedures from the component library in response to end-user requests.

Algorithm Animation Library Manager. The library manager is a process with which client-programmers interact to maintain the component library. The library manager provides a means to create and delete classes and to create, modify, or delete algorithms, input generators, and views within algorithm classes. Although the environment manager uses the component library directly, changes made by the library manager should not affect the environment manager. The modified components should be immediately ready for use by the environment manager.

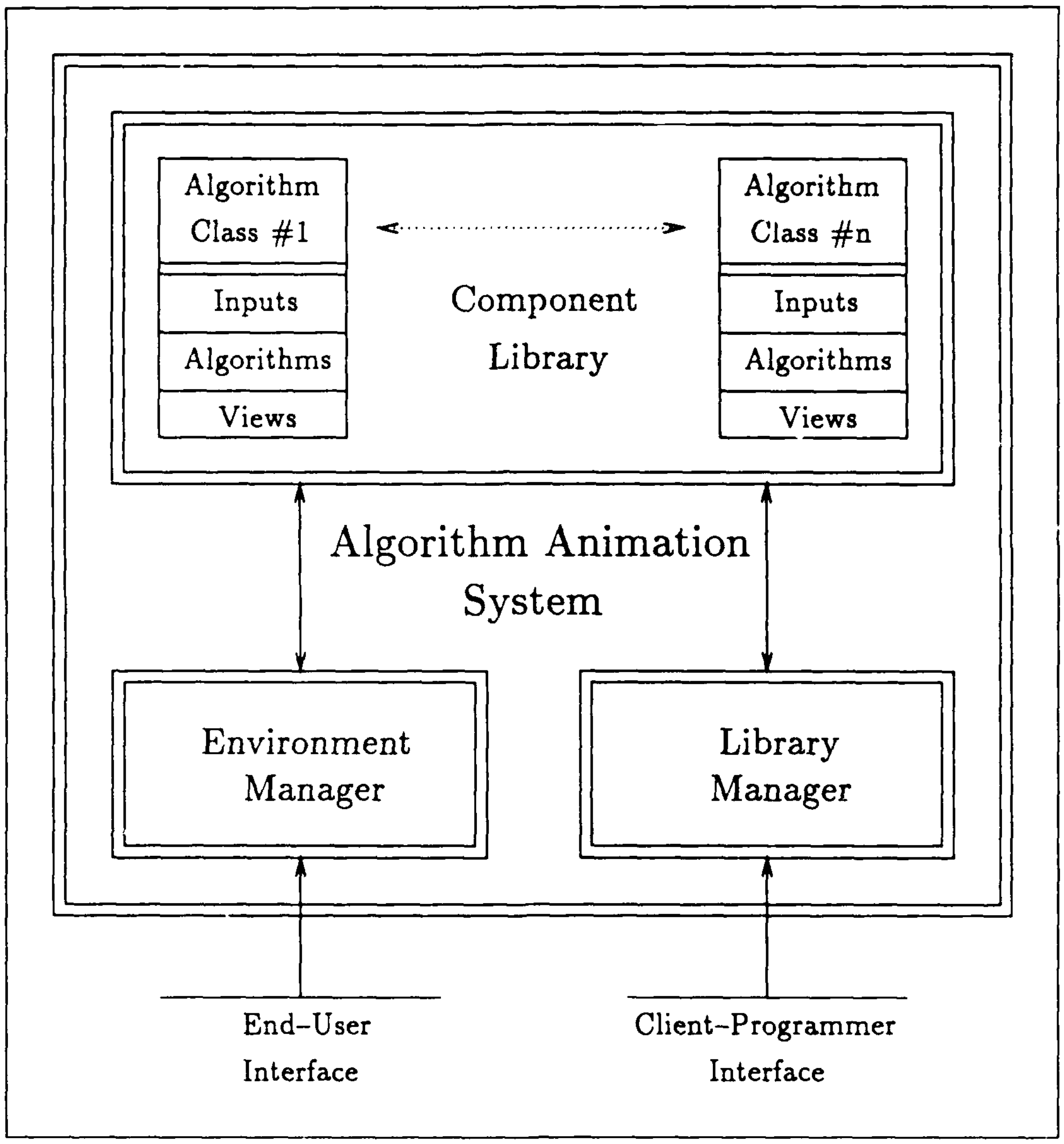


Figure 2.5. An Algorithm Animation System

2.3 Summary of Current Knowledge

Algorithm animation is part of a larger body of research called *program visualization.* Program visualization refers to the use of computer graphics to illustrate some aspect of a program. A related, but distinctly different field of study, visual programming, refers to the use of computer graphics to develop or specify a program. There has been increased interest in both areas of research recently, as shown by the addition of the *Introduction to Visual Programming (and Program Visualization) Environments* course [12] at the ACM SIGGRAPH/89 16th Annual Conference on Computer Graphics and Interactive Techniques. This section surveys previous program visualization research that has influenced this investigation.

Taxonomy of Program Visualization. Brad Meyers[20] classifies program visualization systems by whether they illustrate processes or data, and whether they are static or dynamic. Figure 2.6 shows some program visualization systems and their classifications.

	Static	Dynamic
Process	PegaSys [18] Booch Diagrams [4] Data Flow Diagrams Structure Charts Flowcharts	BALSA [6] PV [5]
Data	Incense [19] Movie/Stills [2]	Sorting Out Sorting [1] BALSA [6] Movie/Stills [2] Animation Kit [17] TANGO [23]

Figure 2.6. Taxonomy of Program Visualization [20:15]

Static displays of code include flowcharts, data flow diagrams, structure charts, Booch diagrams [4:55], and other techniques that attempt to show data or control flow in a static graphical representation of an algorithm. The PegaSys system straddles the fields of program visualization and visual programming by providing computer graphics to represent program structure and design, while applying syntax and design rules to determine whether or not a program meets its pictorial documentation [18:72].

Systems which produce static displays of data are generally designed to monitor and display the value of program variables. These systems are primarily used for debugging. They do not reveal the fundamental operations of the algorithm. Incense [19] automatically generates static displays of data structures. The displays include curved lines, arrow heads, stacked boxes, and user-defined structures. The goal was to "make debugging easier by presenting data structures to programmers in a way that they might have drawn them by hand" [20:18].

Systems which dynamically display code typically display the currently executing section of code and use some sort of highlighting to indicate the currently executing instruction or instructions. The PV project at the Computer Corporation of America [5] attempted to use graphics in all phases of software project management by supporting the manipulation of static and dynamic displays of computer systems, manipulation of program and document text, algorithm animation, and reusable components. Funding for the project was cut just as the implementation of the prototype was beginning [6:31].

Systems which dynamically display data are referred to as algorithm animation systems. The next section briefly describes the most significant research efforts in this area.

Algorithm Animation Systems. In the late '60s and early '70s, graphics hardware was scarce and expensive. While the opportunity to use existing hardware was small, its potential – especially for educators – was great. The obvious solution was to record computer-generated images on film [6:27].

The first systems developed explicitly for algorithm animation were built in the mid-70's at the University of Toronto by Ron Baecker [6:34]. The systems were not interactive; the movies were created by recording frame by frame snapshots of the executing algorithm. While the movies were technically and aesthetically impressive, they required an extremely large amount of time and effort to produce; Baecker's classic algorithm movie, "Sorting out Sorting" [1], took nearly three years to make [6:29].

With the advent of affordable graphics-based workstations, interactive systems have replaced movies for algorithm animation. Using a computer to do real-time animation is now cheaper and faster than making an algorithm movie. Plus, a user can interact with the computer animation, adding a dimension that can never be achieved with movies.

Movie/Stills. At Bell labs, Bentley and Kernighan developed a UNIX-based set of algorithm animation tools for debugging programs, developing new programs, and communicating information about how programs work. "The output is crude, but the system is easy to use; novice users can animate a program in a couple of hours. The system produces movies on Teletype 5620 terminals and Sun workstations and also renders movies into 'stills' that can be included in `troff` documents" [2:abstract]

Animation Kit. At Tektronix, London and Duisberg used the object-oriented programming language, Smalltalk, to animate algorithms. They annotated the algorithms with "interesting events" and used Smalltalk's Model-View-Controller (MVC) system to notify views of an event. They experienced some problems with the MVC system, namely with incremental updating of the displays, compositing multiple views, and back-mapping of the view to the model [17:68-71].

BALSA. The most powerful and flexible algorithm animation system to date is Brown's and Sedgewick's Brown University Algorithm Simulator and Animator (BALSA) system [6]. BALSA, built in the early 80's, was designed to animate, in real-time, any and all algorithms. It is best described as bringing the movie *Sorting out Sorting* to life [6:39]. In his dissertation [6] and in several journal articles [7, 8, 9], Brown describes how BALSA was implemented, outlines the general methods of algorithm animation, and gives examples of BALSA animations. Brown's dissertation presents a thorough and informative history of algorithm animation and provides detailed descriptions of most of the systems mentioned here.

TANGO. John Stasko's Transition-based ANimation GeneratiOn (TANGO) system [23] introduced a design methodology for simplifying the creation of algorithm animation. TANGO provides the client-programmer a small but powerful set of primitive animation commands. The primitives provide a consistent animation design interface, independent of the underlying hardware and software. Stasko's work is significant in that it advances the accessibility of algorithm animation and provides a means for creating animations that exhibit smooth continuous movement.

Applications. One of the first uses for algorithm animation was education – teaching the concepts of control and data structures. Baecker's algorithm movies and Brown's BALSA system had education as their primary goal. At Brown University, courses in programming, algorithms, data structures, and mathematics have been taught successfully with BALSA [9:29]. "... Instructors in the courses were able to cover material at a much accelerated rate (estimated at 30% faster) ... Unfortunately, no controlled experiments could be done ..., the introduction of the lab coincided with the introduction of a new text ..." [6:170].

Another application area is algorithm research and development. A BALSA animation of Knuth's dynamic Huffman trees revealed strange behavior with a particular set of input. Eventually a new, improved algorithm for dynamic Huffman trees was developed. Likewise, new versions of shell sort and merge sort have been developed on the BALSA system [6:5].

System debugging and performance monitoring is another area in which algorithm animation can be useful. Nearly all program visualization systems which produce static displays of program data are used for debugging or performance monitoring. Algorithm animation systems can perform the same function. However, for monitoring data, systems that produce static representations may be more efficient; they can typically generate graphical displays of data automatically based on the data type, with little or no assistance from the end-users. Brown discusses systems programming, but admits that before BALSA can be useful, it must provide some method for automatically creating graphical representations [9:29].

2.4 Requirements

This section presents the requirements analysis for an interactive algorithm animation system. Using algorithm animation system model (Figure 2.5) as a starting point, the analysis is conducted from two points of view: the end-user and the client-programmer.

Specification Technique. Based on the results of the literature review and the rapid prototype, the enumerated requirements specification was compiled. The Air Force Materials Lab's IDEF$_0$ structured analysis language was used to document the high-level requirements analysis. IDEF$_0$ is a version of SofTech's Structured Analysis and Design Technique (SADT) used by the Air Force and other DoD agencies [14]. IDEF$_0$ is described briefly here. The high-level IDEF$_0$ documents for the analysis are included at the end of this section.

The idea behind structured analysis is to reduce the complexity of a problem by hierarchically decomposing the problem into pieces that can be more easily understood. The decomposition can be based on data or processes; IDEF$_0$ is based on the analysis of processes or *activities.* The decomposition is reflected through a series of *function diagrams*, and further documented through the use of a *facing page text* for each diagram. Each functional diagram illustrates one level of the decomposition. Figure 2.7 shows the highest-level IDEF$_0$ diagram for the algorithm animation system. The facing page text provides additional information that is not easily inferred from the diagram. *Data dictionary entries* provide even more detailed information regarding each activity and data item [13:4-5].

The primary object of decomposition in IDEF$_0$ is a process, or activity; for example, *Manage Algorithm Animation System* in Figure 2.7. An activity is represented by a rectangular box on the function diagram. The activity name and number appear in the box. The activity number provides a means for tracing through the hierarchical decomposition.

Interfaces between activities are represented by arrows entering or leaving activity boxes. The arrows represent data produced by or needed by a activity. The type of interface is reflected by the position of the arrow with respect to the box: *input* arrows enter the left side of the activity box, *output* arrows leave the right side of the box, *control* arrows enter the top of the box, and *mechanism* arrows either enter or leave the bottom of the box. The activity is viewed as transforming its inputs into outputs under the guidance of its controls. Control and output arrows are required for every activity; input and mechanism arrows are optional.

End-User Requirements. The end-user of the algorithm animation system should be able to customize the animation environment. Multiple algorithm windows, with multiple views of the algorithm in execution should be possible. The user should control the position and size of algorithm windows and views. As appropriate, the user should be able to zoom in on interesting aspects of an animation and pan through the animation. The user can close algorithm windows at any time. Likewise

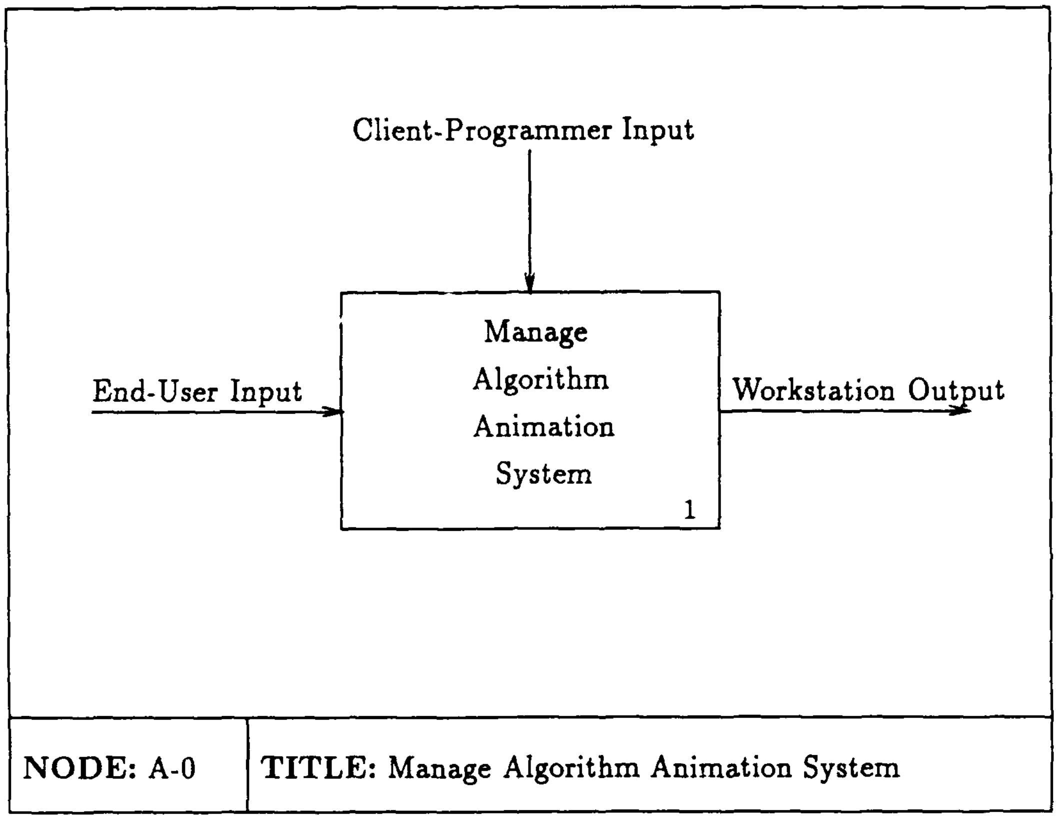

Client-Programmer Input
End-User Input
Manage Algorithm Animation System
1
Workstation Output
NODE: A-0
TITLE: Manage Algorithm Animation System

Figure 2.7. Top-Level $IDEF_0$ Diagram for an Algorithm Animation System

an algorithm window's contents may be replaced at any time. The animation system may be exited at any time.

For each algorithm window, the user must select an algorithm class; that is *sorts*, *tree searches*, *differential equations*, etc. Within an algorithm class, the user may select one of several possible algorithms. For each algorithm, one or more views may be selected which are appropriate for algorithms of that class. Likewise, an input generator is selected. The user may modify algorithm, input generator, and view parameters.

The user may control each animation window individually or control all windows simultaneously. In either case, controls available to the user are: start or continue an animation, restart or reset an animation, pause or terminate the animation, single-step the animation, control the speed of the animation, and set break-points for pausing the animation. Break-point setting and single-stepping are based on an algorithm's interesting events. Since algorithms from different classes may be controlled simultaneously – neither break points nor single-stepping are required mechanisms for simultaneous control.

Some algorithms cannot execute quickly enough to provide an interesting real-time animation. For this type of algorithm, the user needs an *animation recorder*. Within an algorithm window, the animation recorder is used as a video tape recorder. Animations may be recorded either on-line or off-line and played back later. The animation may be viewed at any speed in forward or reverse.

Algorithm windows must provide a mechanism for monitoring and modifying the input parameters, control parameters, algorithm parameters, and view parameters. In addition to the view windows, every algorithm window should support a *status display* which presents statistics describing the current state of the algorithm. The user should be able to interrogate a particular aspect of the animation and have the results of the interrogation displayed.

The animation environment is very flexible; there are many user-selectable options – position, size, and number of algorithm and view windows, algorithms, input generators, views, and parameters. There must be a way for the user to save and restore a particular environment, where the environment includes all user-selectable options available at a particular time.

To help users understand all the options available, on-line help should be available for every interactive function.

Client-Programmer Requirements. A well-defined, consistent programmer interface is essential to the effectiveness of an algorithm animation system. If new algorithms, views, and input generators cannot easily be added to the system, the system cannot fulfill its purpose. The client-programmer should be able to create, modify, and delete classes within the algorithm animation system; and to create, modify, and delete algorithms, input generators, and views within algorithm classes. There should be some sort of automatic validation of new and modified algorithm animations; for example, insuring that the interface to the animation system is correct, that the interfaces between modules within a class are correct, and that there are no naming ambiguities.

The system should provide a library of functions common to all algorithm animations as well as a library of primitives to support color animation. The system should be structured such that the algorithm animations are independent and separate from the main animation controller. Individual algorithm animations can be added, deleted, and modified from the algorithm animation system with no effect on other algorithm animations or the system as a whole. Likewise, as long as the interface to the algorithm animations is not modified, the main algorithm animation controller can be modified without affecting the algorithm animations.

2.5 Summary

This chapter presented the requirements analysis for the development of an algorithm animation system. Section 2.2 discussed the terms and concepts associated with data structures, algorithms, and algorithm animation. The algorithm animation system model and the concept of an algorithm specification were presented. Section 2.3 presented the results of a review of literature relating to algorithm animation and program visualization. Based on the literature review the details of the requirements analysis were presented in Section 2.4. The remainder of this chapter presents the high-level IDEF$_0$ diagrams (Figure 2.8 to Figure 2.12)and the enumerated requirements specification (Figure 2.13 to Figure 2.15).

The next three chapters use the requirements developed in this chapter as the basis for designing, implementing, and testing an algorithm animation system.

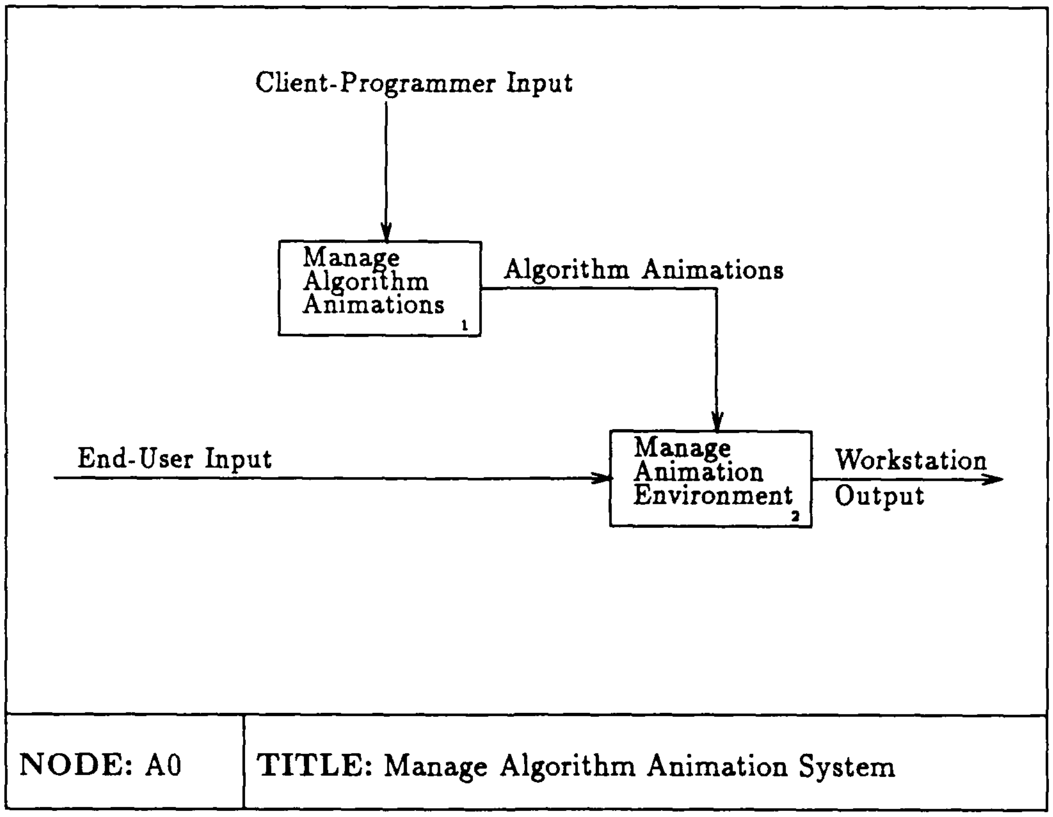

Client-Programmer Input
Manage Algorithm Animations 1
Algorithm Animations
End-User Input
Manage Animation Environment 2
Workstation Output
NODE: A0
TITLE: Manage Algorithm Animation System

Figure 2.8. IDEF$_0$ A0 Diagram - Manage Algorithm Animation System

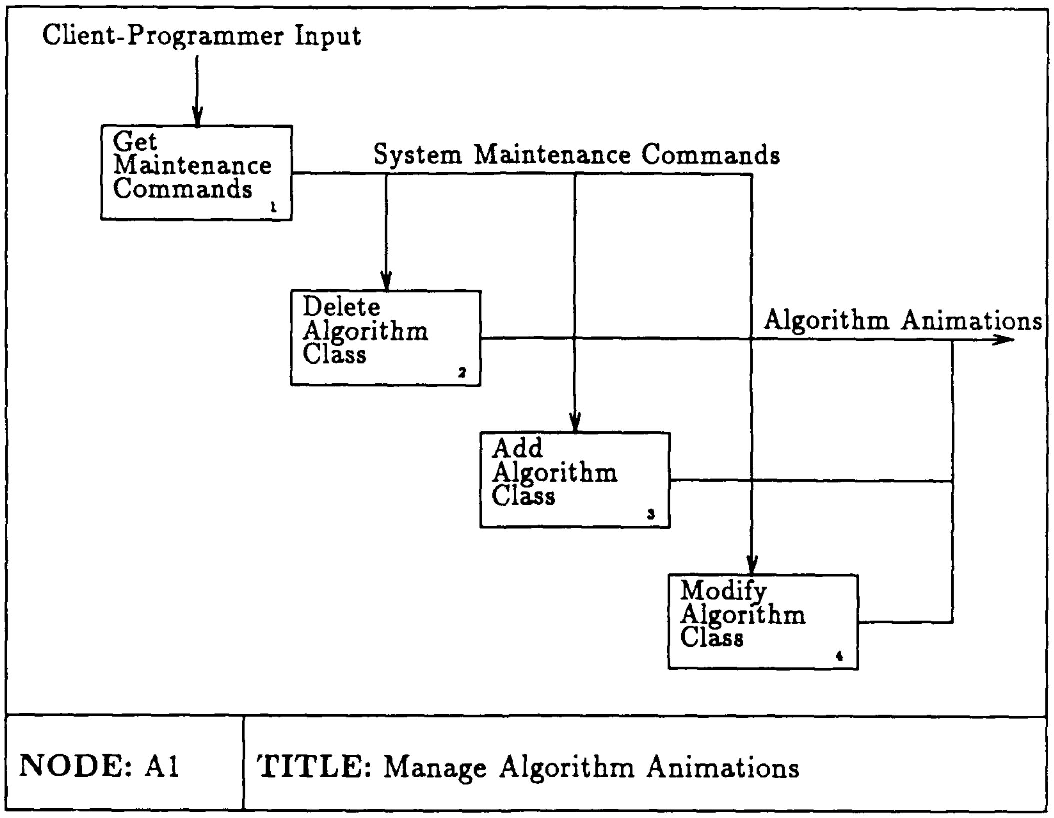


Figure 2.9. IDEF$_0$ A1 Diagram - Manage Algorithm Animations

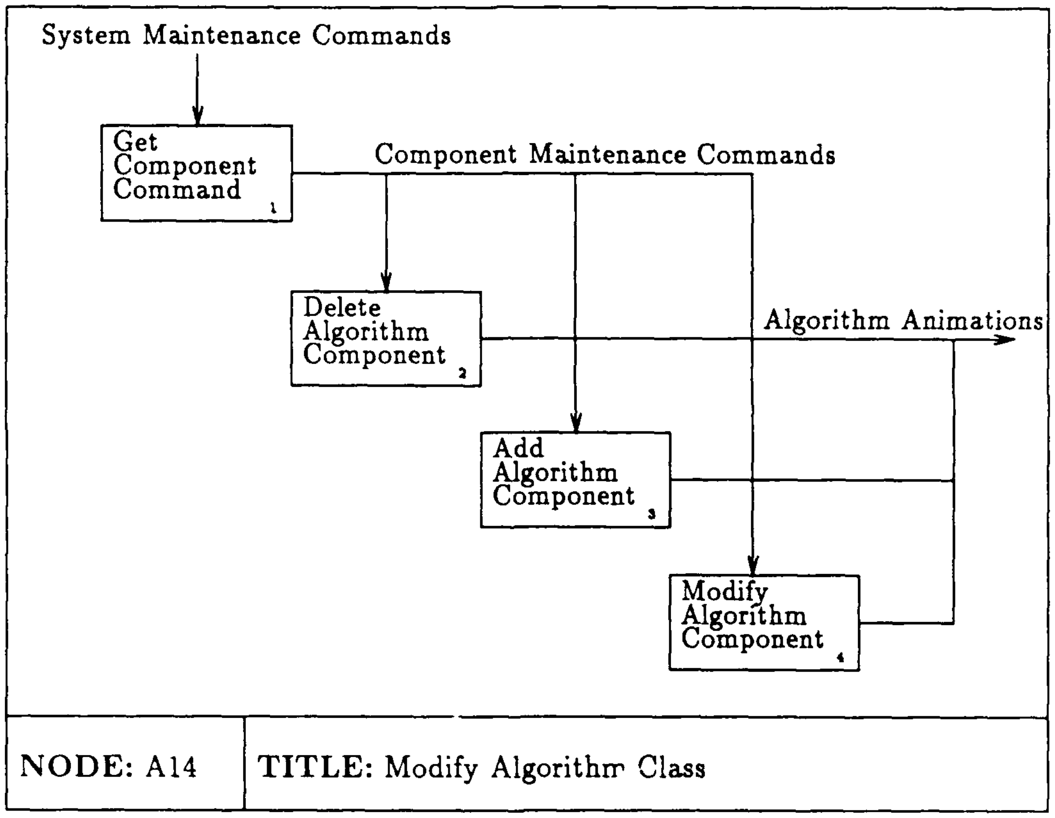

System Maintenance Commands
Get Component Command 1
Component Maintenance Commands
Delete Algorithm Component 2
Algorithm Animations
Add Algorithm Component 3
Modify Algorithm Component 4
NODE: A14
TITLE: Modify Algorithm Class

Figure 2.10. IDEF$_0$ A14 Diagram - Modify Algorithm Class

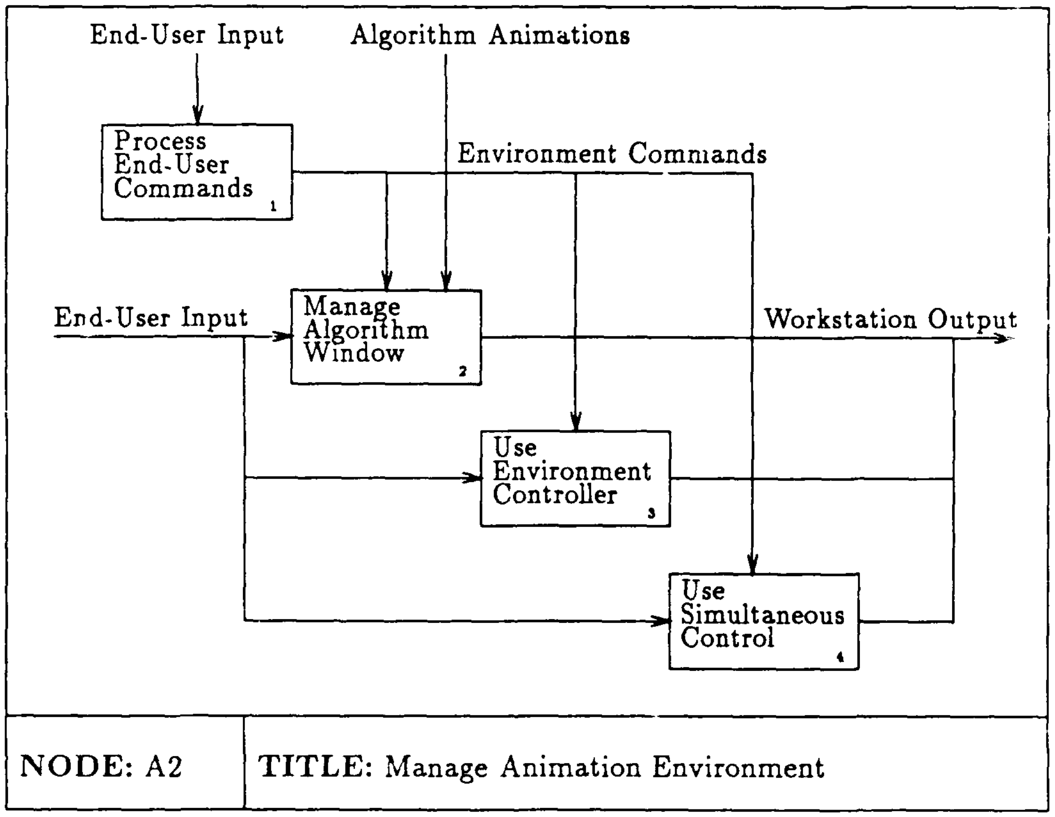

End-User Input
Algorithm Animations
Process End-User Commands 1
Environment Commands
End-User Input
Manage Algorithm Window 2
Workstation Output
Use Environment Controller 3
Use Simultaneous Control 4
NODE: A2
TITLE: Manage Animation Environment

Figure 2.11. IDEF$_0$ A2 Diagram - Manage Algorithm Environment

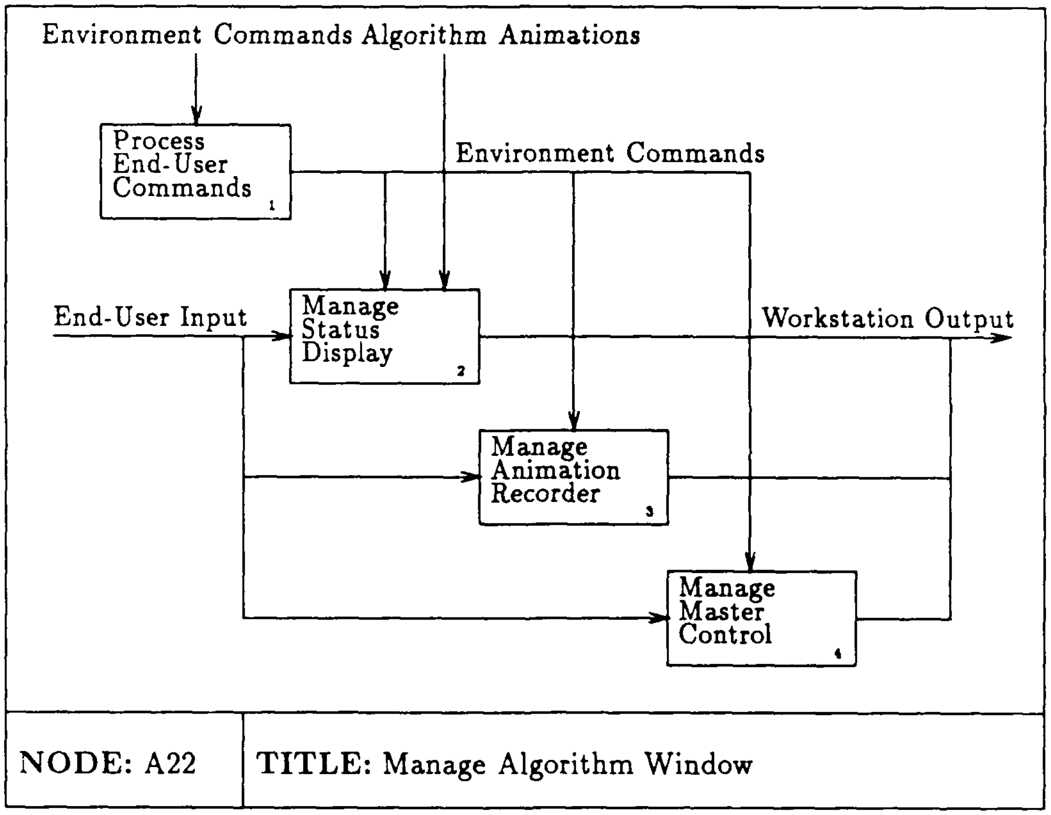


Figure 2.12. IDEF$_0$ A22 Diagram - Manage Algorithm Window

- 1.0 Establish a user interface to the algorithm animation environment manager
  - 1.1 Allow user to customize the algorithm animation environment
    - 1.11 Provide multiple algorithm windows
    - 1.12 Provide multiple view windows within each algorithm window
    - 1.13 Allow user to position and size algorithm and view windows as desired
    - 1.14 Allow user to zoom and pan display in view windows at any time
    - 1.15 Terminate animation session at any time
  - 1.2 Allow user to select algorithms, inputs, and views for each algorithm window
    - 1.21 Select algorithm class from a list of available classes
    - 1.22 Select algorithms for animation from a list of available algorithms within a selected class
    - 1.23 Specify control parameters for an algorithm from a list of control parameters associated with a selected algorithm
    - 1.24 Specify input parameters for an algorithm from a list of input parameters associated with a selected class
    - 1.25 Select views parameters for an algorithm from a list of views associated parameters with a selected class
    - 1.26 Close animation window at any time

Figure 2.13. Enumerated Requirements for Algorithm Animation System (1 of 3)

1.3 Allow user to control execution of algorithm animations

- 1.31 Select control mode: individual, simultaneous
- 1.32 Begin (reset) an animation
- 1.33 Control speed of animation
- 1.34 Terminate (pause) animation at any time
- 1.35 Restart a paused (terminated) animation
- 1.36 Run animation in single-step mode
- 1.37 Set break-point for pausing animation; based on algorithm events for the particular algorithm

1.4 Provide a central control mode

1.5 Provide an animation recorder mode

- 1.51 Start recording all algorithm interesting events generated by an animation within a view window
- 1.52 Stop animation recording
- 1.53 Save animation recording
- 1.54 Load animation recording
- 1.55 Play animation recording forward or backward at any speed
- 1.56 Provide option to turn off display during recording session
- 1.57 Provide option to record animation off-line

1.6 Provide environment save and restore function

- 1.61 Save all user selections currently in effect
- 1.62 Restore previously saved environment (overrides current environment)

1.7 Provide on-line help for every interactive function

Figure 2.14. Enumerated Requirements for Algorithm Animation System (2 of 3)

- 2.0 Establish a programmer interface to the algorithm animation library
  - 2.1 Allow programmer to create, modify, and delete algorithm animations within the algorithm animation system
    - 2.11 Create, modify, and delete algorithms
    - 2.12 Create, modify, and delete algorithm views
    - 2.13 Create, modify, and delete algorithm input generators
  - 2.2 Provide automatic validation of changes to algorithm animation system
    - 2.21 Interface errors are reported immediately
    - 2.22 Protect algorithm animation system from accidental corruption during development of new algorithm animations
  - 2.3 Provide library of primitive functions to support color animation.
  - 2.4 Provide library of functions common to all algorithm animations.
  - 2.5 An individual algorithm class can be developed or modified without any effect on other algorithm classes or the algorithm animation system as a whole.

Figure 2.15. Enumerated Requirements for Algorithm Animation System (3 of 3)

III. System Design

3.1 Introduction

This chapter presents a preliminary software design for an algorithm animation system – The AFIT Algorithm Animation Research Facility (AAARF). Throughout the remainder of the investigation, AAARF refers to this particular design and implementation of an algorithm animation system. Preliminary design is concerned with the transformation of requirements into a software architecture. Chapter IV presents the detailed design which focuses on refinements of the architectural representation leading to detailed data structures and algorithmic representations for implementation [21:215].

An *object-oriented* design methodology was used to translate the requirements into high-level *objects* and *operations.* An object-oriented approach was particularly well-suited because an algorithm animation system consists of individual elements (objects) with which end-users and client-programmers interact; data flow is at best a side issue. This chapter describes the test methodology, the object-oriented design technique, and the AAARF high-level objects.

3.2 Test Methodology

Two commonly used software testing approaches are "black box" and "white box" testing [21:470]. These are generic classifications; within each category there are several specific testing methods.

Black box testing methods focus on the functional requirements of the software. These methods attempt to find errors in the following categories:

1. Incorrect or missing functions,
2. Interface errors,
3. Errors in data structures,
4. Performance errors, and
5. Initialization and termination errors [21:484].

White box testing methods are based on the close examination of procedural detail. They strive to

1. Exercise all independent paths within a module at least once,
2. Exercise all logical decision for both the TRUE and FALSE cases,
3. Execute all loops at their boundaries and within their operational bounds, and
4. Exercise all data structures to assure their validity [21:472].

No white box testing is possible at this point in the design; it is deferred until the detailed design and implementation phase. However, some black box testing is possible. Basically the requirements specification is compared to the objects and operations defined in the design to insure that all requirements are met. The object interfaces and data structures are also inspected. Further black box testing is conducted during the detailed design and implementation stage (Chapter IV).

3.3 Object-Oriented Design

Object-oriented design (OOD) is a relatively new design concept that has evolved over the past 15 years [21:335]. OOD builds upon four important software design concepts: ***abstraction***, ***information hiding***, ***reusability***, and ***modularity***. This section discusses the terms and procedures associated with OOD.

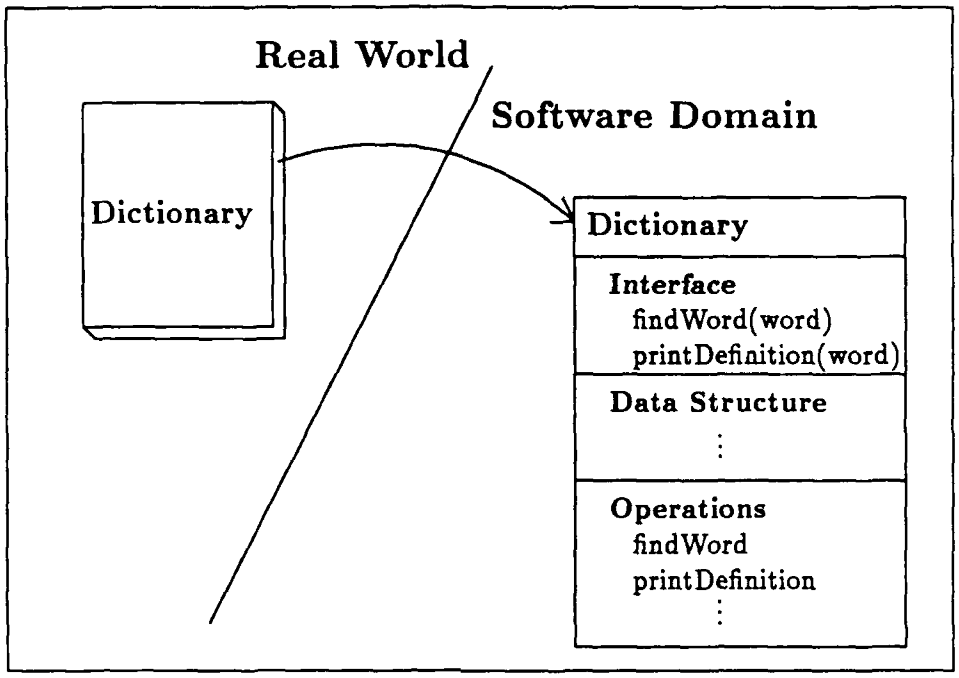


Figure 3.1. Mapping a Real World Object into the Software Domain [21:337]

Objects, Operations, and Messages. An *object* is an abstract data type (Section 2.2) that represents a real world entity in the software domain (Figure 3.1). An object consists of a *private part* and a *public part.*

The private part consists of the underlying data structure that represents the object's state and a set of *operations*, or *methods*, for manipulating the data structure. The public part is the object's interface to the rest of the software system; it consists of *messages* which specify what operation is to be performed on the object, but not *how* the operation is to be performed [21:336].

Defining a private part and providing messages to invoke appropriate processing achieves information hiding - the details of implementation are hidden from all program elements outside the object. Objects and their operations provide inherent modularity - software elements (objects and operations) are grouped with a

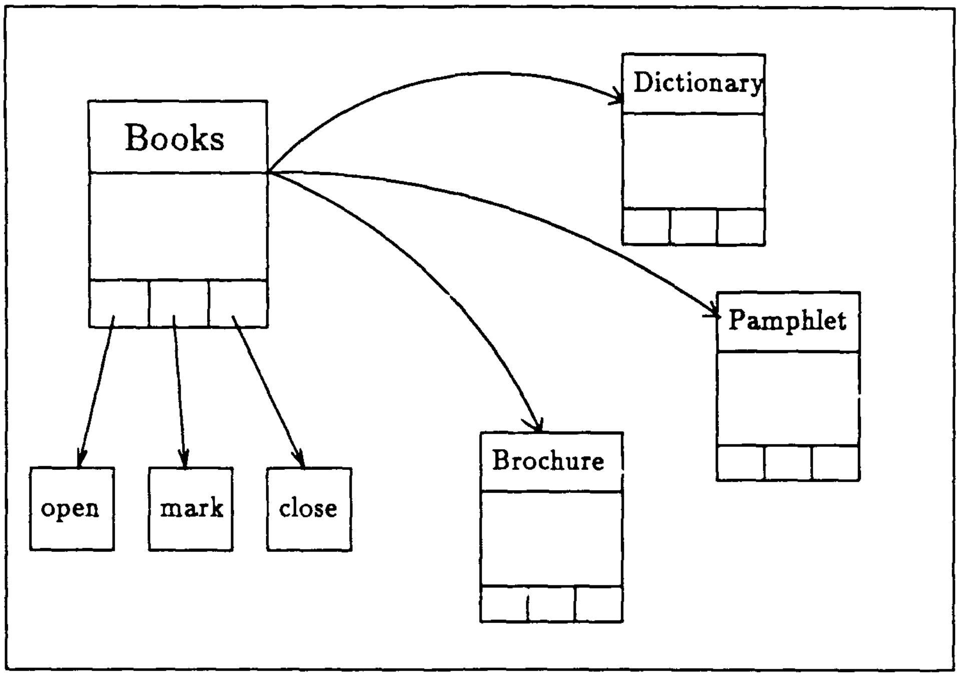


Figure 3.2. Objects of the Class *Books* [21:338]

well-defined interface (messages) to the rest of the software system [21:336].

Classes, Instances, and Inheritance. A *class* is a set of objects that have the same or similar characteristics and perform the same or similar operations. An *object* is an *instance* of a larger class [21:338]. All objects are members of a *class* and *inherit* the private part of the class.

For example in Figure 3.2, the object *Dictionary* is an instance of the larger class *Books* and inherits the operations *open*, *close*, and *mark*. *Books* is a data abstraction that makes creating new instances a simple matter [21:338].

The software engineering goal of reusability is achieved by creating objects that build on existing attributes and operations inherited from the class. Only the new object's differences from the class must be specified, rather than defining all the characteristics of the new object [21:338].

Object-Oriented Design Method. Booch suggests the following procedure for OOD:

- Identify the Objects. Establish classes, objects, and instances.
- Identify the Operations. Determine the operations that may be meaningfully performed on the objects or by the objects.
- Establish the Visibility. Determine relationships between objects; what other objects are seen by a given object.
- Establish the Interface. Establish object interfaces (messages). The interface forms the boundary between the outside (public) view and the inside (private) view of the object.
- Implement Each Object. Choose a suitable representation and implement the interface. [4:48-49]

3.4 High-Level AAARF Design

This section presents a high-level OOD design for AAARF. Two high-level graphical object classes and an algorithm component object class were defined for the design. The names and operations of the two graphical object classes coincidentally map almost directly to objects provided by SunView. At this level of design the correspondence is of no consequence, but during the detailed design and implementation phase the coincidence proves to be very convenient.

In this section, the high-level object classes and their interfaces are defined. Specific object instances and their relationships are also described. The abstract data type (ADT) specification notation of Figure 2.1 is used to define the AAARF objects. The axioms are omitted; they are used primarily for proving the correctness of a process and that is beyond the scope of this study.

The window *Object Class.* A window is an abstraction that represents a rectangular region of the workstation display. Text or graphics can be displayed in a window. windows can be displayed within other windows. windows can be moved and resized. As shown later in this section, certain types (instantiations) of windows may limit the basic window attributes. Figure 3.3 lists the ADT specification for the window object class.

```
structure window(Pos, Size, Text, Graphics)
    declare   create() → window
              show(window) → window
              hide(window) → window
              destroy(window) → window
              setPosition(window, Pos) → window
              getPosition(window) → Pos
              setSize(window, Size) → window
              getSize(window) → Size
              putText(window, Text) → window
              putGraphics(window, Graphics) → window
    end
end window
```

Figure 3.3. Specification for AAARF window Object

AAARF Window Hierarchy. The AAARF design incorporates three levels of windows to provide the user with multiple simultaneous algorithm animations and multiple views of each animation. Figure 3.4 shows the instances of window used by AAARF and their relationship to one another.

Main Window. This is a window within which all interaction with AAARF is contained. The main window supports multiple instances of *algorithm windows.* No animation actually takes place in the main window; it serves as a high-level manager of the animation environment.

Algorithm Window. Like the main window, algorithm windows exist mainly to contain other windows. Every algorithm window is associated with a particular algorithm class. Algorithm windows may be created and destroyed at any time, but there is limit to the number that can exist at any given time (an implementation detail). Each algorithm window supports multiple instances of *view windows.* Algorithm windows can be resized and moved anywhere on an AAARF main window, and they may overlap one another.

View Window. Animations are displayed in view windows. Every view window is associated with a particular view of an algorithm. At least one view window is active within every algorithm window (the maximum is left as an implementation issue). View windows can be resized and moved, but they cannot extend beyond their algorithm window, and they may not overlap. This forces a modular animation environment and minimizes the end-user's opportunity to "loose" windows.

The panel *Subclass.* The panel is a subclass of window. panels provide a means for the end-user to monitor and modify system parameters. panels cannot be resized, but retain all other characteristics of the window class. The basic window operations are supplemented with operations to manipulate low-level panelItem objects which provide the interface to the user. panelItems can be associated with particular action operations which are invoked in response to a panelItem state change.

One set of panel objects is associated with the *main window* and another with each instance of *algorithm window.* Figure 3.6 and Figure 3.9 show the relationship between windows and their panels.

- Main window panels
  - Environment Panel. The environment panel is used to save, restore, and delete animation environments.

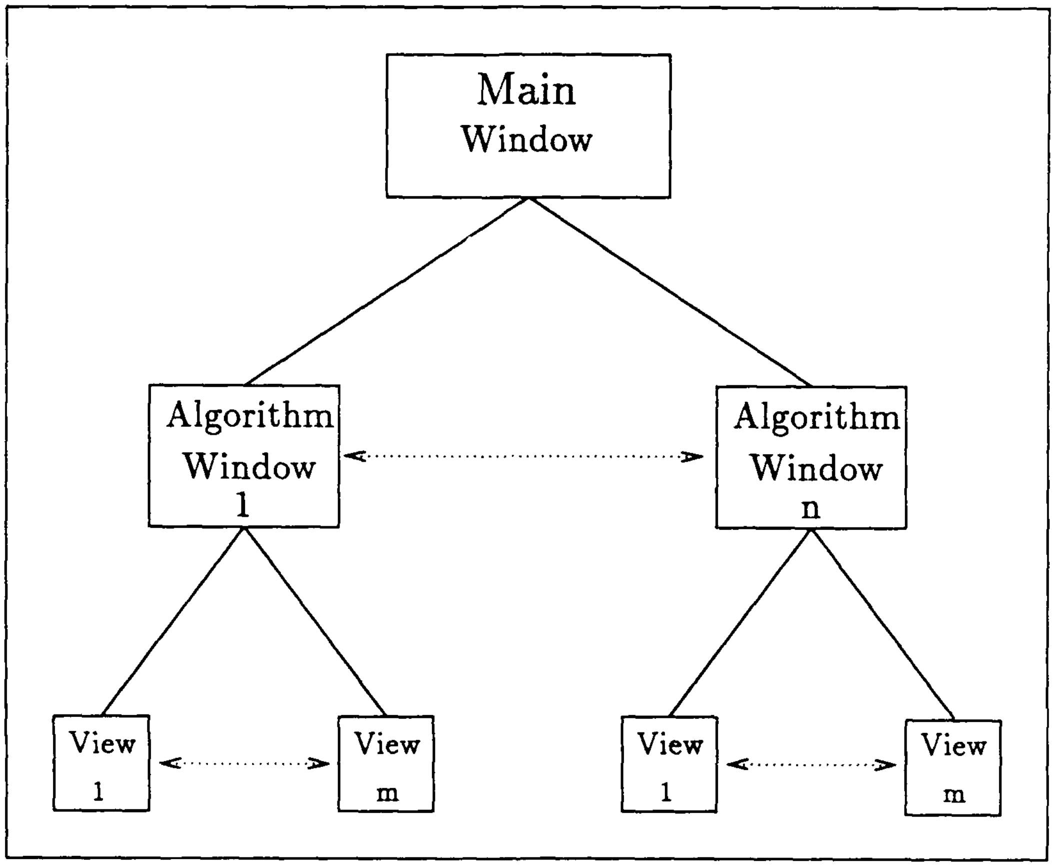


Figure 3.4. Instances of `window`

  - Control Panel. The control panel is used to impose simultaneous control on all active algorithm windows.

- Algorithm window `panels`

  - Master Control Panel. The master control panel provides a way for the user to modify the input, view, algorithm, and control parameters for an animation.
  - Recorder Panel. The recorder panel allows the user to record and playback animation recordings; and to save, load, and delete recordings.

– Status Panel. The status panel provides the user with information regarding the current state of the animation or some particular aspect of the animation.

The menu *Object Class.* The menu provides a method for the user to make a selection from several options. menus consist of low-level objects called menuItems. menuItems can be associated with a particular operation which is invoked when the menuItem is selected. Figure 3.5 describes the menu class operations.

```
structure menu(Pos, noItems, menuItem, itemNo, Selection)
     declare   create() → menu
               show(menu) → menu
               hide(menu) → menu
               destroy(menu) → menu
               getSelection(menu) → Selection
               setPosition(menu, Pos) → menu
               setMenuItem(menu, menuItem) → menu
               getMenuItem(menu, menuItem) → itemNo
               deleteMenuItem(menu, menuItem) → menu
               getNoItems(menu) → noItems
     end
end menu
```

Figure 3.5. Specification for AAARF menu Object

The AAARF design uses two instances of menu to provide the user with high-level decisions on two levels.

Main Menu. This menu is used to select a new algorithm animation and to activate panels for simultaneous control of animations and for saving and restoring animation environments.

Algorithm Window Menu. An algorithm window menu is associated with every algorithm window. This menu is used to activate various panels which control the animation, record the animation, and present a status display for the animation.

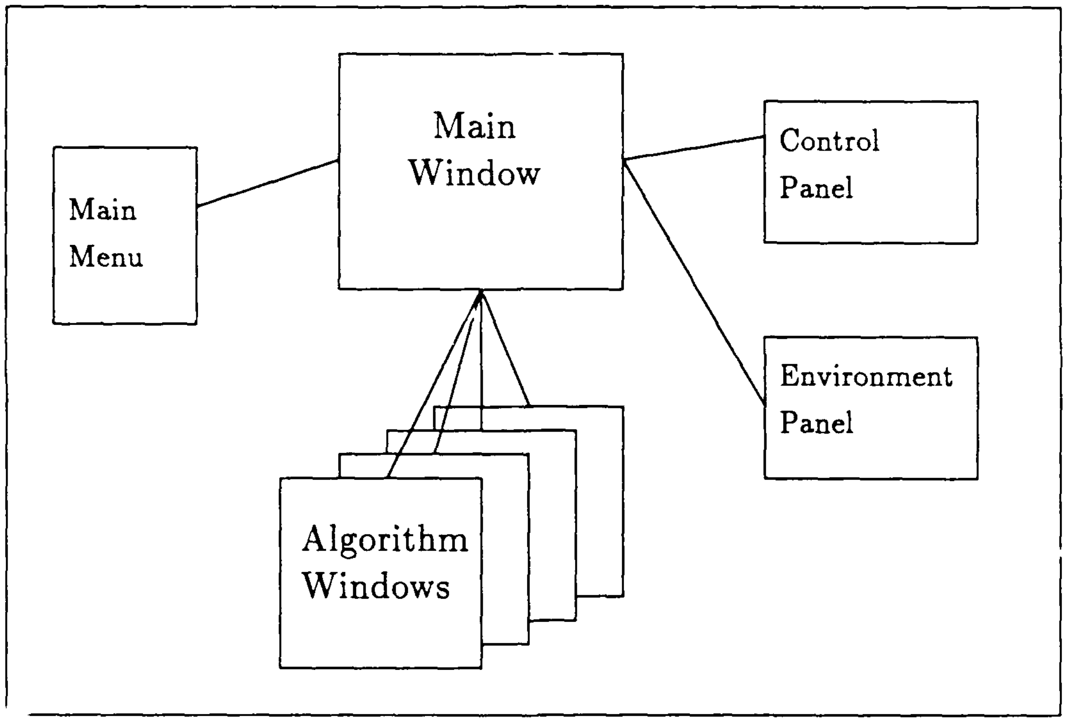


Figure 3.6. Structure of the Main Window

The component *Object Class.* The component object provides a means to select and control the algorithm components depicted in Figure 2.4. The component objects are associated with algorithm windows and view windows, but not with the main window. Figure 3.7 show the operations associated with the component object class.

```
structure component(componentName, Parameter, Value)
    declare  set(component,componentName) → component
             get(component) → componentName
             setParameter(component, Parameter, Value) → component
             getParameter(component, Parameter) → Value
    end
end window
```

Figure 3.7. Specification for AAARF component Object

The AAARF design uses three types of components: *input*, *algorithm*, and *view*. Each type supplements the basic operations supplied by the class with a set of component-specific operations. Figure 3.8 shows the additional operations for each type of component object. Figure 3.9 illustrates how the *components* fit into the algorithm window structure.

Object: input component getInput(dataStructure)

Object: algorithm component
getIE(interestingEventStructure)
executeAlgorithm()

Object: view component
processIE(interestingEventStructure, viewParameters)
updateView(viewParameters)
paintView(viewParameters)

Figure 3.8. Supplemental Operations for AAARF component Objects

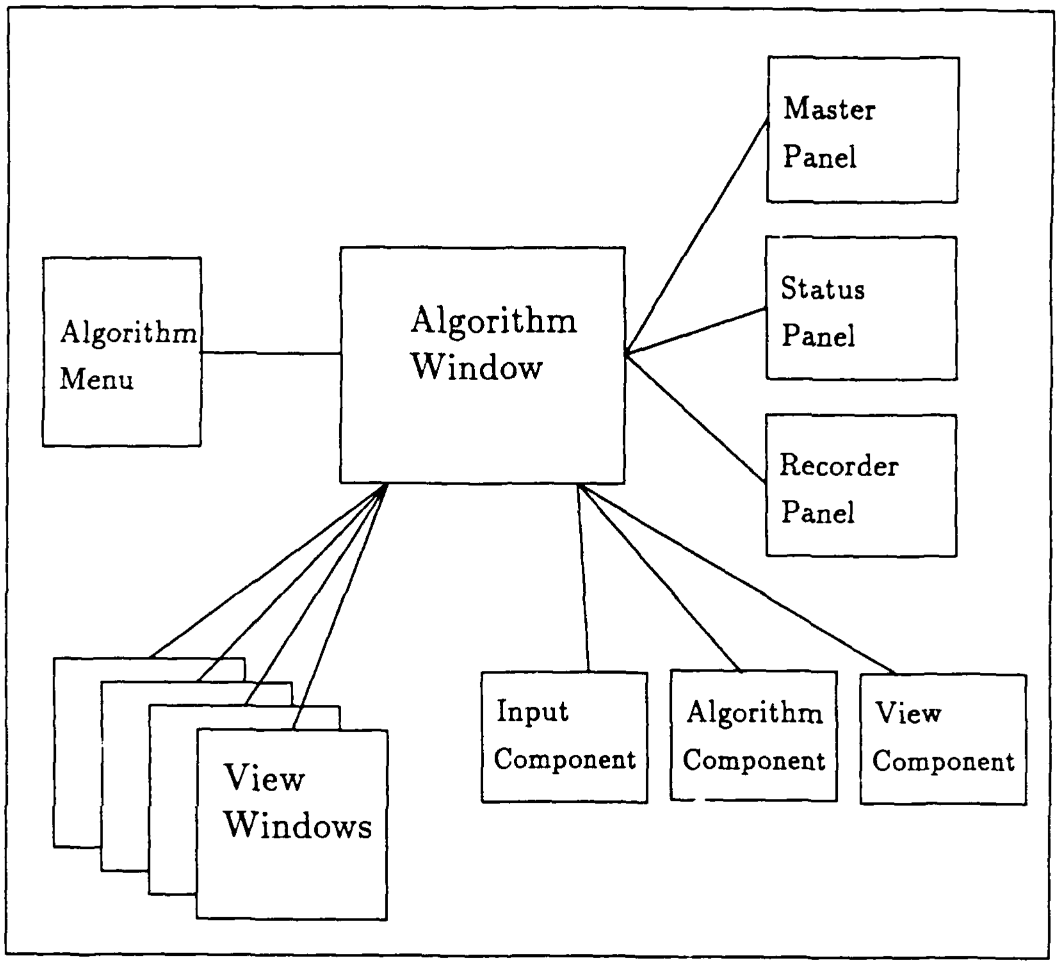


Figure 3.9. Structure of Algorithm Windows

3.5 Summary

This chapter described the object-oriented design technique, the design objects and operations, and the test methodology for the design. In the next chapter, the preliminary design is translated into the detailed design and implementation of AAARF.

IV. Detailed Design and Implementation

4.1 Introduction

This chapter discusses the detailed design and implementation of an algorithm animation system on a Sun4 workstation using the SunView window-based user interface environment. This particular implementation is called the AFIT Algorithm Animation Research Facility (AAARF) and is based on the preliminary design developed in Chapter III. As with any design implementation, this is just one of many possible realizations.

The topics discussed in this chapter include: the detailed design methodology, the implementation approach, the testing methodology, SunView, and major implementation decisions. Appendix A and *The AAARF Programmer's Guide* in Appendix D provide additional details of the implementation. For even finer detail, see the AAARF source code in Appendix E; it is included as Volume 2.

4.2 Detailed Design Methodology

The preliminary design (Chapter III) provided an object-oriented architectural representation for AAARF. Operations, interfaces, and data structures were defined for the objects that comprise the AAARF design. The detailed design is concerned with developing an algorithmic abstraction for coordinating the activities of the AAARF objects. In this stage of design, the preliminary design objects and operations are used to develop a high-level procedural description of the software.

Structure charts are used to represent the (hierarchical) procedural structure of the AAARF program modules. Structure charts are an effective program structure notation, yet simple enough that practically anyone can understand them with little or no explanation. Other program structure notations, such as Warnier-Orr and

Jackson diagrams, could be used with equal effectiveness [21:210]. Appendix A contains the structure charts for the AAARF implementation.

4.3 Implementation Approach

Typically, the first step in the implementation stage is determining the target system, the programming language, and the system tools for implementing a particular design. Leaving these decisions until the implementation phase prohibits characteristics of a programming language or the target hardware architecture from influencing the software design and data structures.

In this investigation, it was determined from the outset that AAARF would execute on Sun workstations. During the course of the design development, the SunView window-based environment was selected as the basis of the AAARF implementation; SunView provides nearly all the graphical objects specified in the AAARF preliminary design. SunView is described in Section 4.5. C was selected as the programming language. It provides the speed required for computer graphics intensive applications, and supports modern software engineering concepts. The Sun workstation operating system, SunOS, and SunView provide an excellent interface to C.

Having selected the programming language, target machine, and program development tools, the AAARF detailed design modules (Section 4.2) are mapped to program modules. Many of the AAARF objects map directly to SunView objects, significantly simplifying the implementation. Several important implementation decisions are made: how to handle multiple algorithms and views, how to link algorithm classes to the main algorithm animation system, and how to help the client-programmer develop new animations. The highlights of this step are described in Section 4.6. Finally, the implementation is tested as described in Section 4.4.

4.4 Implementation Testing

Implementation testing is conducted on three levels: unit testing, integration testing, and validation testing [21:502]. These are the white box testing methods mentioned in Section 3.2. Unit testing is applied to individual functions as they are developed. After unit testing, integration testing is performed on the complete program to ensure the interfaces between functions are correct. Finally, validation testing is performed to provide final assurance that the program meets the stated requirements.

In this study, the detailed design is implemented using a top-down integration approach [21:507]. The high-level module interfaces are developed first; stubs are used to exercise the function interfaces. Most stubs do little or no data manipulation. As stubs develop into bona fide procedures, unit testing is performed. Integration and unit testing are conducted repeatedly throughout the development process.

Validation testing is the continuation of the black box testing begun in Section 3.2. In these final tests, AAARF is tested against the requirement specification to insure that all requirements are met. "*Alpha testing and beta testing* are conducted to uncover the errors that only an end-user can find. Alpha testing is done at the developers site with the developer 'looking over the shoulder' of the user, and beta testing is done at the users site without the developer"[21:515]. In this investigation, alpha testing is conducted in the AFIT graphics lab with several student and instructor end-users. Beta testing is discussed in Chapter 5.3.

4.5 SunView

SunView is an object-oriented system that provides a set of visual building blocks for assembling user interfaces. SunView objects include windows, pointers, icons, menus, alerts, panel items, and scrollbars. It also provides a *notification-based* input system. AAARF makes extensive use of SunView objects and the SunView Notifier.

SunView Objects. The highest level object class in SunView is the *Object* (a rather confusing name for an object class). *Window* is a subclass of *Object*, and *Subwindow* a subclass of *Window*. Figure 4.1 depicts the SunView object class hierarchy and instances of each class. The following SunView object descriptions are taken from the *SunView Programmer's Guide* [28].

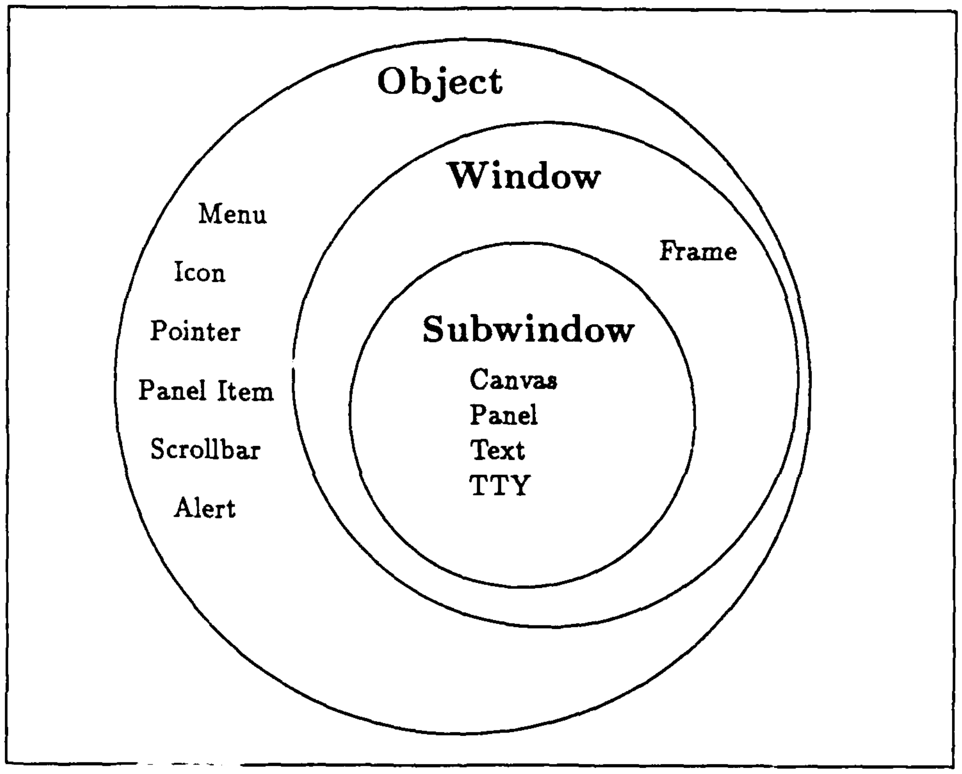


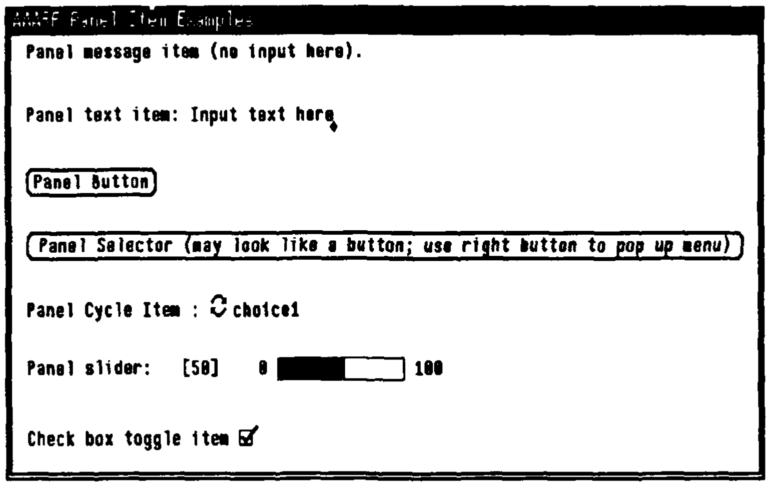
Figure 4.1. SunView Objects [28:10]

Window Objects. *Window* objects include *Frames* and *Subwindows. Frames* contain non-overlapping *Subwindows* within their borders. There are four types of *Subwindows* provided by SunView:

- *Panel Subwindow* — A *Subwindow* containing *Panel Items.*
- *Text Subwindow* — A *Subwindow* containing text.
- *Canvas Subwindow* — A *Subwindow* into which programs can draw.
- *TTY Subwindow* — A terminal emulator, in which commands can be given and programs executed.

Other Visual Objects. The other types of objects, like *Windows*, are displayed on the screen, but they differ from *Windows* in that they are less general and more tailored to their specific function. They include:

- *Panel Item* — A component of a *Panel* that facilitates a particular type of interaction between the user and the application. There are several types of *Panel Items*, including *buttons*, *message items*, *choice items*, *text items*, and *sliders*. Figure 4.2 shows some examples of SunView panel items.
- *Scrollbar* — An object attached to and displayed within a *Subwindow* through which a user can control which portion of the *Subwindow's* contents are displayed.
- *Menu* — An object through which a user makes choices and issues commands. *Menus* pop up when the user presses the right mouse button.
- *Alert* — A rectangular region on the screen which informs the user of some conditions. It has one or more buttons which the user can push to dismiss the *Alert* or choose a means of continuing. Figure 4.3 shows an example of a SunView alert.
- *Pointer* — The object indicating the mouse location on the screen.
- *Icon* — A small image that represents the SunView application on the SunView work screen. [28:11-12]


AAARF Panel Item Examples

Panel message item (no input here).

Panel text item: Input text here

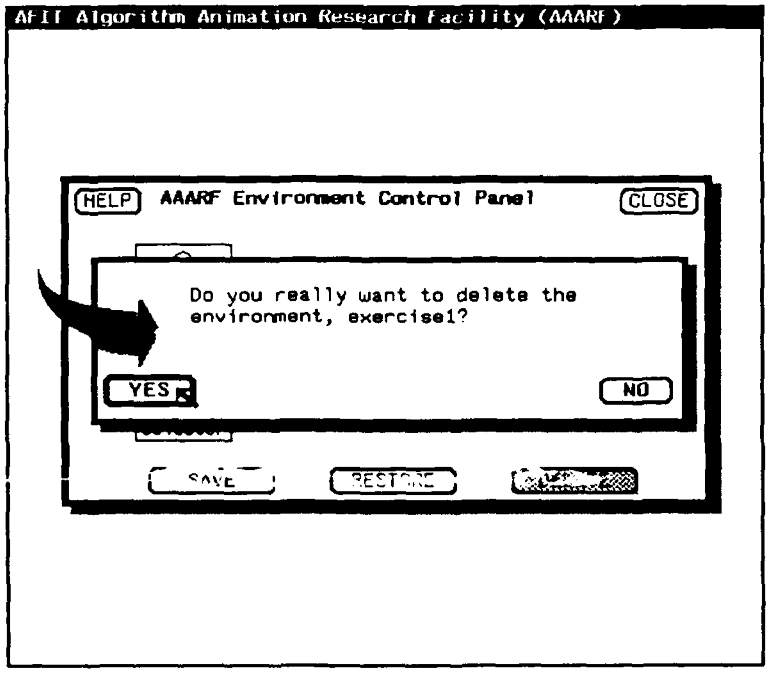
Panel Selector (may look like a button; use right button to pop up menu)

Panel Cycle Item : choice1

Panel slider: [50] 0 100

Check box toggle item

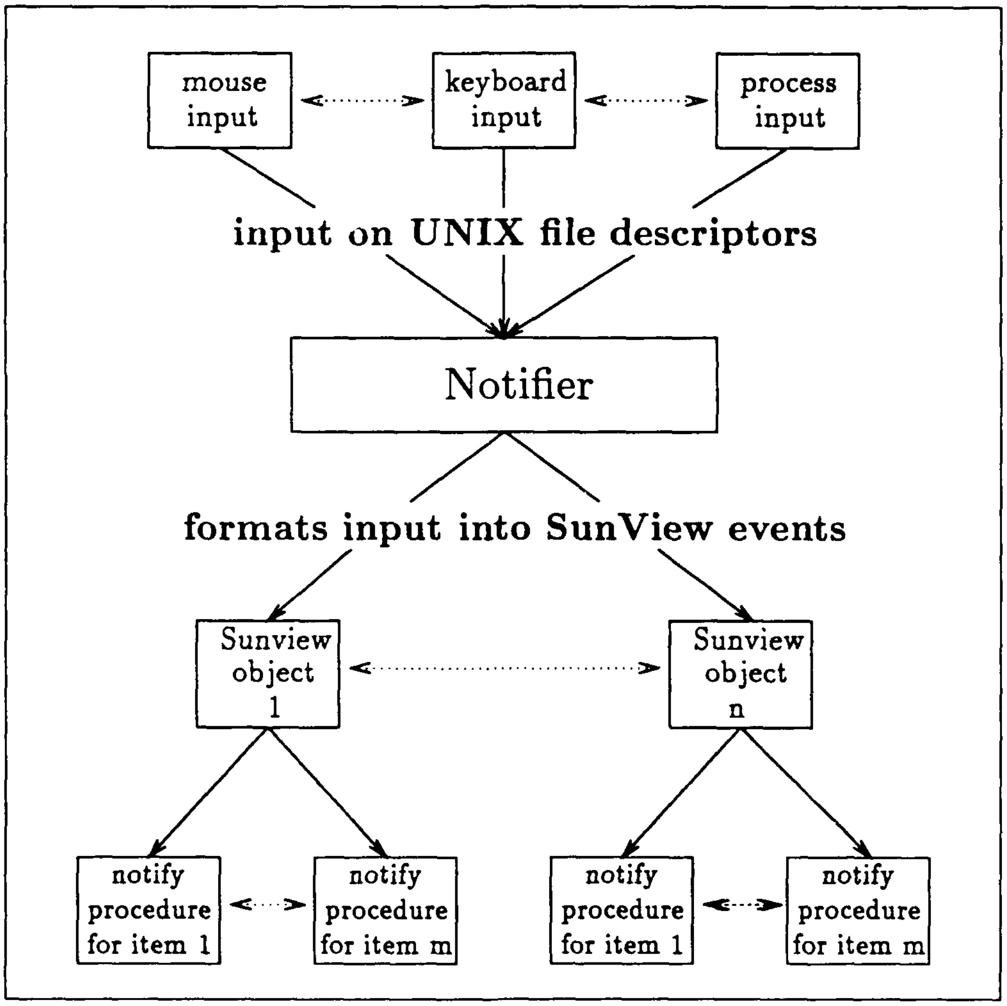
Figure 4.2. SunView Panel Item Examples


AFIT Algorithm Animation Research Facility (AAARF)

Do you really want to delete the environment, exercise1?

Figure 4.3. SunView Alert Example

The SunView Notifier. SunView is a *notification-based* system. The Notifier acts as the controlling entity within an application, reading UNIX input from the kernel, and formatting it into higher-level *events*, which it distributes to the appropriate SunView objects. The Notifier *notifies*, or calls, various procedures which the application has previously registered with the Notifier [28:20]. These procedures are called *notify procedures.* The SunView Notifier model is shown in Figure 4.4.



mouse input
keyboard input
process input
input on UNIX file descriptors
Notifier
formats input into SunView events
Sunview object 1
Sunview object n
notify procedure for item 1
notify procedure for item m
notify procedure for item 1
notify procedure for item m

Figure 4.4. SunView Notifier Model [28:23]

4.6 Implementation Decisions

One of the first major implementation decisions is *how should the multiple algorithm classes be managed.* Another issue involves determining *what can be done to help the client-programmer develop new algorithm animations.* These and related decisions are addressed in this section.

Managing Algorithm Classes Figure 2.5 shows an idealistic abstract model of an algorithm animation system in which a *library manager* provides an interface through which client-programmers manage algorithm animation components, and an *environment manager* provides an interface through which end-users select algorithm components to create interesting algorithm animations. In software, the concept of a component library is extremely complicated and difficult to implement. Component compatibility and dependencies have to be maintained and checked every time the end-user makes a selection.

A more practical implementation approach is to develop an algorithm class library. Each element of the library is an independent algorithm class process, consisting of the input generators, algorithms, and views associated with the algorithm class. This approach eliminates the need for compatibility and dependency checking by taking advantage of the following characteristics of algorithm classes:

- Algorithms in the same algorithm class operate on identical data structures,
- A view that can be used with one algorithm can be used with nearly any other algorithm from the same algorithm class. This says nothing about the effectiveness of the view.
- An input generator that can be used with one algorithm can be used with any other algorithm from the same algorithm class.
- *Most* input generators and views are not shared across algorithm classes.

There remains the problem of how to incorporate the algorithm class library into the algorithm animation system. By design, the algorithm class library is a dynamic entity; it's envisioned that client-programmers will add new algorithm classes to AAARF indefinitely. Three possible solutions are proposed:

1. Link the algorithm class library to the main animation system to produce a single AAARF executable program. Two obvious problems with this solution are (1) the size of the executable image is a function of the size of the library and (2) even if some sort of dynamic linking [27:52-53] is used to reduce the size of the executable image, the system has to be compiled and linked every time a change is made to an algorithm component. This situation becomes very complicated when two programmers are developing and testing two different algorithm animations simultaneously.
2. Develop some method to dynamically load algorithm class object files from the algorithm class library and link to them while the AAARF program is executing. This is similar to the first solution except that in this case, the main program is not linked with the algorithm class until execution time. This solution eliminates the need to compile and link every time an algorithm component is modified and permits parallel development of algorithm animations. However, it's difficult to implement and entails some overhead for managing the execution-time loading and linking.
3. Let the algorithm class library consist of independent executable algorithm class programs which are invoked by a central animation environment manager as separate processes and controlled via some form of interprocess communication (IPC) [25:192-200]. This solution creates one executable image for each algorithm class plus a high-level animation manager program. It provides for parallel development of algorithm animations and is simple to implement. This approach is selected for implementation.

The third approach provides a flexible animation environment for the end-user and a manageable development environment for the client-programmer. AAARF consists of a central process that invokes and controls multiple algorithm class processes. The central process is referred to as the main process, the environment manager, or AAARF depending on the context. The algorithm class processes are referred to as algorithm processes or class-specific processes. The algorithm processes are essentially stand-alone except for the IPC hooks to the main process.

UNIX sockets [25:191-217] provide the IPC facilities for the main AAARF procedure to control algorithm class processes. The UNIX IPC interface makes IPC similar to file I/O. A process has a set of I/O descriptors for reading and writing. The descriptor may refer to files, devices, or communications channels.

Managing Algorithm Components Given that algorithm classes are implemented as separate processes with IPC hooks to the main AAARF process, *how should an algorithm process be implemented?* The problem is that both the algorithm component and the view component require access to the data structure being manipulated by the algorithm. In addition, the algorithm component needs some mechanism for reporting interesting events to the view component. There must also be a mechanism for controlling (starting, stopping, breaking, etc) the animation.

The algorithm animation component model (Figure 2.3) suggests a solution. The component model forms a pipeline which drives the graphic representation displayed on the view windows. The input component drives the algorithm component; the algorithm component drives the view component; and the view component generates the graphics commands. Figure 4.5 shows a slightly modified version of the algorithm animation component model. In this version, the view component requests and receives interesting events from an abstract interesting event generator. The interesting event generator may be an algorithm component, a *software recording* of interesting events, or an unknown source on a remote computer system.

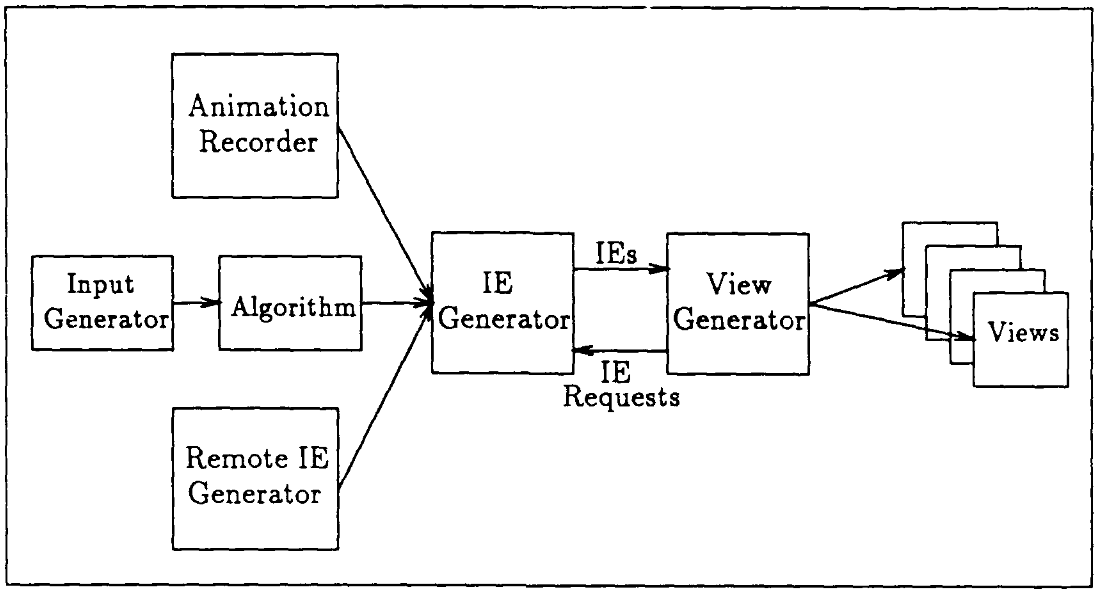


Figure 4.5. Extended Algorithm Animation Component Model

AAARF algorithm processes are composed of two levels: an interesting event generator and a view generator. These two levels are implemented as separate processes. Both processes require access to the same data structure. This can be handled in a couple of different ways:

- Shared memory — both processes access the data structure. This method requires some contention and data locking consideration to protect the data from accidental corruption through simultaneous access by both processes.
- IPC — each process keeps a copy of the data structure. The interesting event generator determines what action is taken on the data structure and informs the view generator via socket-based IPC. This method is easier to implement and makes it easy to animate algorithms that are running on remote hosts.

The view generator process consists of the view component and is referred to as the class-specific window-based process. The interesting event generator process consists of the input and algorithm components and is referred to as the class-specific background process. The background process generates interesting events for the window-based process. Socket-based IPC is used to request and report interesting events and to exchange parameters. By associating a polling timer with the view component, the SunView Notifier calls the view component at appropriate intervals to update the animation. The animation is controlled through the view component.

AAARF Architecture Summary. This section summarizes the results of the preceding sections. AAARF uses three levels of execution linked via UNIX sockets to implement the algorithm animation system model. Figure 4.6 shows the levels of execution.

The AAARF Main Process. AAARF's top-level process acts as the environment manager. It provides the main screen, a mechanism for saving and restoring animation environments, a mechanism for controlling multiple algorithm animations simultaneously, and a means for starting new algorithm animations.

The Class-Specific Window-Based Process. The middle level of execution is the class-specific window-based process. The window-based level of execution provides the animation views, an animation recorder, a status display for interrogating the state of the algorithm, and a master control panel for monitoring and modifying the parameters that affect the animation. The view component of Figure 2.5 is implemented at this level.

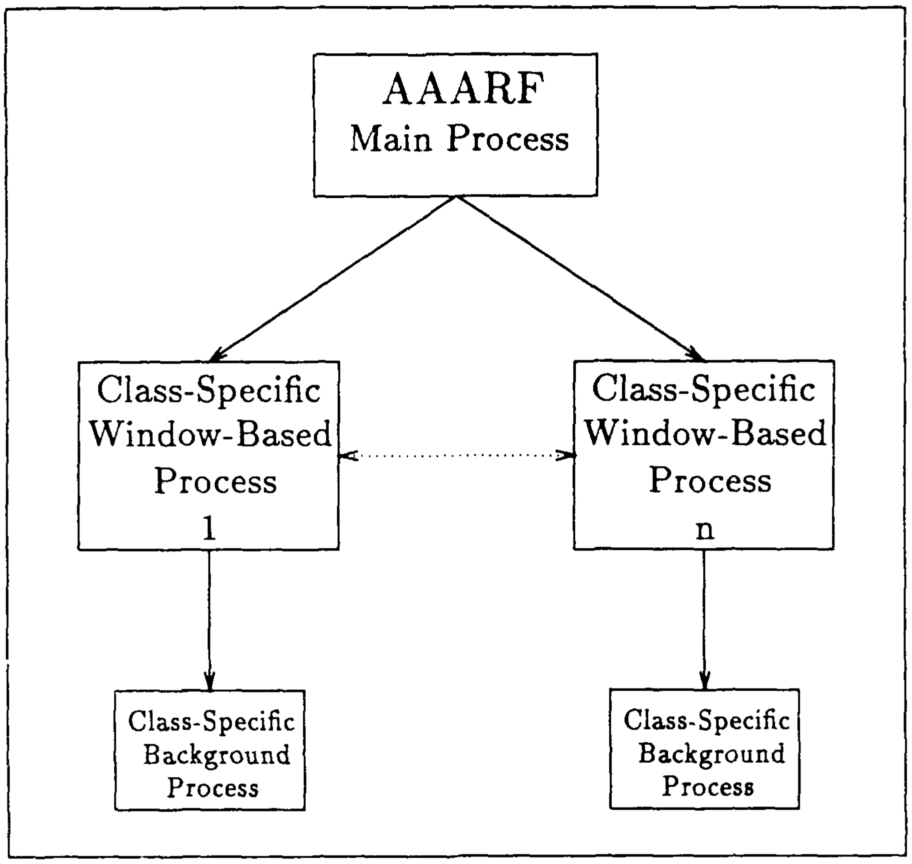


Figure 4.6. AAARF Levels of Execution

The Class-Specific Background Process. The lowest level of execution is transparent to the user; this level implements the input generator and algorithm components of the algorithm animation system model (Figure 2.5). It provides the window-based level with the IEs that drive the animation. The background process waits with an IE until the window-based level sends an IE request. The background process sends the IE, then resumes execution of the algorithm. At the next IE, the background process again stops and waits for an IE request from the window-based process. Figure 4.7 shows the background process model.

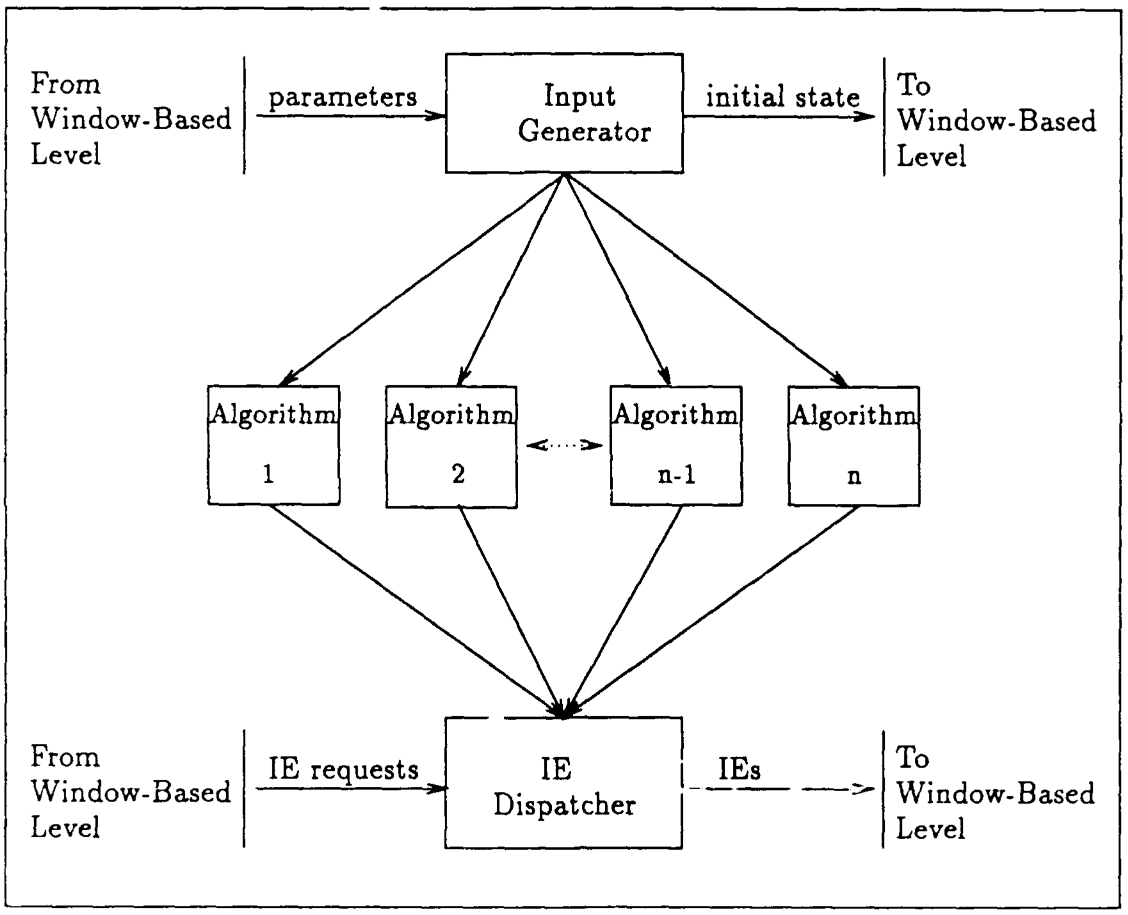


Figure 4.7. Background Process Structure

AAARF Class-Common Library. Each algorithm class that AAARF invokes is actually a stand alone process. Without the communication links to AAARF, the process could be executed without the AAARF main process. Every algorithm class process is characterized by the following initialization sequence:

- Get setup parameters from AAARF,
- Create base window for animation,
- Create the master control panel,
- Create the algorithm window menu,
- Create the animation canvases (views),
- Create the animation recorder,
- Create the status display,
- Run the background process,
- Paint the active canvases,
- Enter the SunView main loop.

The AAARF class-common library presents client-programmers a framework for developing new algorithm animations by providing all the common functions. The client-programmer simply develops the class-specific input, algorithm, view, and control functions. Because the class-common library depends on some class-specific data structures, linkable object code is not provided, only source code. The class-common framework must be recompiled for each new algorithm class. This limitation could be eliminated in a future version of AAARF.

Not only does the AAARF class-common library make animation development easier for client-programmers, it also forces a consistent animation interface for the end-user. Every algorithm class supports the same set of tools, and those tools work identically for every algorithm class. After animating one algorithm, end-users can animate any algorithm, regardless of the algorithm class or its client-programmer. The class-common library also ensures a consistent communications interface with the AAARF main process.

Appendix D describes the process of creating new algorithm animation in some detail. It explains what is provided by the class-common library and what is required from the client-programmer.

Program Documentation. The existing source code is fully documented in accordance with AFIT System Development Documentation Guidelines and Standards [14]. Most functions are less than one page long and most modules contain less than ten functions.

4.7 Summary

This chapter described the AAARF detailed design and implementation. Implementation essentially consisted of the top-down mapping of AAARF design objects to their corresponding SunView objects. The *AAARF Programmer's Guide* (Appendix D) and the *AAARF User's Manual* (Appendix C) discuss the implementation and operation of AAARF in greater detail. Both documents include many illustrations of actual AAARF screen images.

The test methodology, design technique, and implementation approach were also discussed. Important implementation decisions were presented in detail. The implementation relies heavily on the SunView window-based environment; the Sunview objects and Notifier were discussed. In the next chapter AAARF is evaluated, and recommendations for further research are offered.

V. Conclusions and Recommendations

5.1 Introduction

This chapter presents the results of the AAARF system testing and an evaluation of AAARF's functionality and educational value. Based on the test results and evaluation, conclusions and directions for future research are presented.

5.2 System Testing

AAARF passed all aspects of white box testing (Section 3.2); every function operates as designed. There are no known programming errors. As for black box testing, not every requirement from the enumerated requirements specification was implemented. The remainder of this section discusses each of the unimplemented requirements.

Pan and Zoom. The pan and zoom features for view windows (Requirement 1.14) were not implemented. With the SunView *Scrollbar* object, these capabilities should be easy to add. However, not all animations benefit from the pan and zoom capabilities; either the end-user or the client-programmer must decide if a particular view should support the capabilities. The simplest solution is to require that all animations support panning and zooming, then the decision to display the pan and zoom controls (presumably *Scrollbars*) is left to the user.

Reverse Playback of Animations. The animation recorder reverse playback feature (Requirement 1.55) was not implemented. The recorder panel shows a reverse playback button, but hitting the button reveals a message explaining that the feature is not currently available.

Running an algorithm in reverse is no simple matter. As an animation progresses, every interesting event affects a unique change to the algorithm state structure, q:

$$q_k \xrightarrow{IE} q_{k+1}$$

The animation recorder works by adding each new interesting event to the end of a doubly-linked list. The list is doubly-linked ostensibly for traversal in both directions during playback. The recorder flawlessly delivers interesting events in the forward direction, exactly duplicating a previously recorded animation. The animation recorder can also reliably deliver interesting events in reverse order, however interesting events alone cannot transform the current algorithm state to a unique previous state:

$$q_k \overset{IE}{\not\rightarrow} q_{k-1}$$

Animations cannot be played in reverse based solely on interesting events; more information has to be available regarding the algorithm state before an interesting event occurred. The problem is definitely solvable, but implementing the solution required more time than was available, so this feature was left unimplemented.

Automatic Validation of Animation Library. The requirement for automatic validation of changes to the animation library (Requirement 2.2) was not explicitly addressed. Since the *animation library* is actually a distributed collection of executable processes, any subset of which may be accessed by AAARF during a particular session, it is difficult to define, much less validate, *changes* to the library. The idea of an animation library is actually an abstraction of the collection of executable processes; the automatic validation is provided by the development environment tools — the compiler, the linker, *lint*, etc. Whether or not this requirement has been met is arguable; in any case, no specific tools were created to support this requirement.

Library of Animation Primitives. The library of animation primitives (Requirement 2.3) was not implemented. A library of animation primitives is an important step toward simplifying the creation of algorithm animations, but the development of such a library is an undertaking that is beyond the scope of this study. The SunView Pixrect library provides several powerful graphics primitives, but provides nothing to directly support animation.

5.3 Evaluation of AAARF

Satisfying the functional requirements and producing error-free code are certainly desirable goals for any software engineering project, but in the case of AAARF, some important questions remain:

- Can end-users effectively use it?
- Is it of any use to end-users?
- Can client-programmers create new algorithm animations?
- Will it run on other hosts?

The first three questions cannot be completely answered until a substantial number of end-users and client-programmers are surveyed. Such a survey is currently underway as part of the AAARF beta testing. In conjunction with the AFIT's *Advanced Algorithms and Data Structures* course, students are using AAARF to study specific algorithm behavior and to develop new algorithm animations. Based on their experiences with AAARF, the students complete a questionnaire [15] (Appendix B) designed to evaluate AAARF's educational value as well as its user-interface and functionality.

No formal user evaluations have been collected yet. However, several end-users and one client-programmer used AAARF during its alpl a testing. The remainder of this section addresses each question based on observations made during the alpha test period.

Functionality. With few exceptions, AAARF follows the user interface conventions recommended by Sun [28:469-474] and used by nearly all SunView applications. Veteran Sun workstation users can view algorithm animations almost immediately. Even end-users who are unfamiliar with SunView can view simple algorithm animations within three to five minutes of beginning their first session with AAARF.

AAARF offers many options for selecting, controlling, and viewing algorithms. Despite numerous *pop-up* help screens, naive users are occasionally unaware of some system features. The help screens describe every feature that can be accessed from the window, panel, or menu associated with the help message. The user surveys should reveal whether the "naive user problem" is a fault of the system or a characteristic of the user.

Because so many options are available, the AAARF main window can become littered with algorithm windows and their panels — it's possible to have 14 control panels and 16 view windows open simultaneously. Visually managing that much information is difficult to say the least. There is very little automatic window layout. On color systems, algorithm windows and their associated panels are color-coded. On monochrome systems, it is nearly impossible to tell with which algorithm window a particular control panel is associated. The problem is a consequence of the flexibility given to the user. Limiting the flexibility could reduce the potential complexity of the displays, but would stifle the users ability to create unique and interesting animation environments. Familiarity with the system should relieve the problem for most users. The environment control facility can also help with the problem; complex arrangements of windows, panels, and parameters can be saved and restored quickly and with no confusion.

Educational Value. David Scanlan conducted research on learner preferences for graphic or verbal algorithmic presentation techniques and found that "graphical expressions of algorithms seem more helpful to most students than verbal expressions" [22:176]. Considering that 78% of the students he surveyed preferred graphical presentations, his findings seem believable. But, there is some skepticism toward program visualization — Edsger Dijkstra states

> I was recently exposed to a demonstration of what was pretended to be educational software for an introductory programming course. With its "visualizations" on the screen it was such an obvious case of curriculum infantilization that its author should be cited for "contempt of the student body," ... We must expect from that system permanent mental damage for most students exposed to it. [11:xxxvii]

Expert opinion notwithstanding, experience with AAARF has shown that that end-users can learn how an algorithm works from a "good" view of the algorithm in execution. What makes a "good" view varies from one algorithm to the next. Within the *sorts* class, the *tree* view works well with *heap sort*, the *sticks* view with *shaker sort*, and the *dots* view with *quick sort*. Allowing the user to select any of several possible views, permits them to find their own particular "good" view.

Unfortunately, only two algorithm classes have been implemented for AAARF so far — *sorts* and *binary tree traversals*. Based on these two classes it's difficult to determine the general educational value of AAARF. As for the extent of the mental damage to end-users, only time will tell!

Creating Algorithm Animations. The *sorts* class was developed along with the AAARF main process and the class-common library. The *binary tree traversals* class was developed by an independent client-programmer. While one client-programmer success does not provide sufficient grounds for claiming an effective client-programmer interface, it does show that client-programmers can use the class-common library to build algorithm animations. More client-programmers must use AAARF before any conclusions can be drawn regarding the effectiveness of the client-programmer interface.

The *binary tree traversals* class made use of the *tree* view from the *sorts* class. Some modifications were required to reflect the unique nature of the traversal algorithm state representation, but the views are essentially identical. Thus, algorithm components can be reused. As more animations are created, it will be easier for client-programmers to draw from existing views and input generators.

Portability. AAARF was designed to execute on Sun3 and Sun4 workstations. It was developed primarily on a Sun4 workstation. Due to the strong dependency on SunView, it is unlikely that the top two levels of AAARF can execute on any other computer, however the low-level background process can execute on any machine that is reachable with socket-based IPC. AAARF's view procedures are not particular about the source of their interesting events — a local background process, the animation recorder, or a remote system. As long as socket-based IPC is possible, the input generator and algorithms can be controlled and interesting events can be retrieved.

There are some portability problems with AAARF. It requires more resources than some systems are configured to provide. In most cases, the program will run, but with limited capabilities; less than four algorithm windows can be opened simultaneously. It's difficult to determine exact system requirements because other processes running on the system affect resource availability. AAARF's notable resource requirements are in two areas: memory and window devices. Both of these resources are controlled by the system administrator.

- With up to sixteen animation canvases opened simultaneously, AAARF uses a lot of memory. The amount of memory is directly proportional to the size and number of open canvases. Rather than reduce the maximum number of views or algorithm windows, the size of the animation canvas is reduced. The

AAARF administrator sets the window size to the largest size that does not cause the system to run out of memory. A 800 × 800 pixel canvas works for most systems.

- AAARF uses 64 window devices. Since the user normally has several other sunview windows open at least 96, possibly 128, window devices must be installed in the /dev directory. On systems which only have 64 window devices installed, only two algorithm windows can be opened.

5.4 Conclusions

Based on the work performed during this study, this section presents the conclusions reached regarding AAARF and algorithm animation.

Literature Review. The literature review was an essential part of this development. Several previous algorithm animation systems were reviewed and aspects of each influenced AAARF: Baecker's displays from "Sorting out Sorting", BALSA's displays and Brown's input-algorithm-view paradigm for algorithm animation, Kernighan's and Bentley's animation pipeline, London's and Duisberg's object-oriented approach, and TANGO's animation library. It is unlikely that AAARF could have succeeded without the guidance of these previous research efforts.

Object-Oriented Design. The object-oriented design technique worked well for designing an algorithm animation system. The animation system was easily partitionable into *objects*.

SunView. The SunView window-based environment provided an excellent basis from which to implement an algorithm animation system. SunView directly supported all of the high-level graphical objects defined in AAARF's preliminary design.

Sun Workstations. A Sun4 workstation configured with 50 megabytes of swap space and 128 window devices was an outstanding host machine for an algorithm animation system. Other system configurations worked, but usually with limited capabilities.

Color Monitors. Color monitors provide much more informative displays than monochrome monitors. Texture patterns help monochrome monitors to emulate color, but there is no substitute for bright colors when attempting to bring a viewers attention to a particular aspect of a graphical image.

End-User Functionality. End-users can animate algorithms with AAARF. Figure 5.1 shows an AAARF display of four sort algorithms. While only two algorithm classes have been implemented, there is no reason to suspect that an algorithm class exists that cannot be animated.

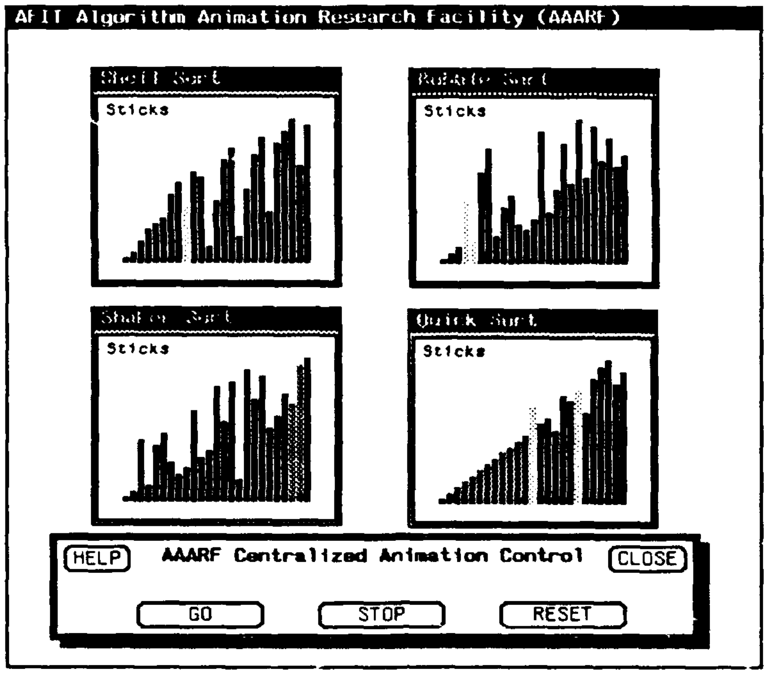


Figure 5.1. Animation of Four Sorting Algorithms

Client-Programmer Functionality. Client-programmers can create new AAARF algorithm animations. Only two classes have been implemented, but the utility of the class-common library has been clearly demonstrated through those implementations.

Teaching Algorithms Through AAARF. Algorithms can be learned from animations of their execution on AAARF. Whether animation is a better method for learning algorithms than conventional methods is a matter for further study.

The Difficulty of Animating Algorithms. Creating algorithm animations is not easy. There are several difficulties:

- Identifying an algorithm's interesting events,
- Selecting an abstract representation for the algorithm state,
- Implementing the abstract representation,
- Presenting the implementation (size, position) on the screen,
- Animating the representation (transitions between images),
- Updating the animation displays (timing, speed),
- Bringing attention to areas of interest.

It is due to these difficulties that automatic animation of algorithms seems unlikely. Figure 5.2 shows four views of a heap sort algorithm; none of the views were generated solely on the basis of changes to the data structure — all of the views require some information regarding the nature of the changes to the data structure.

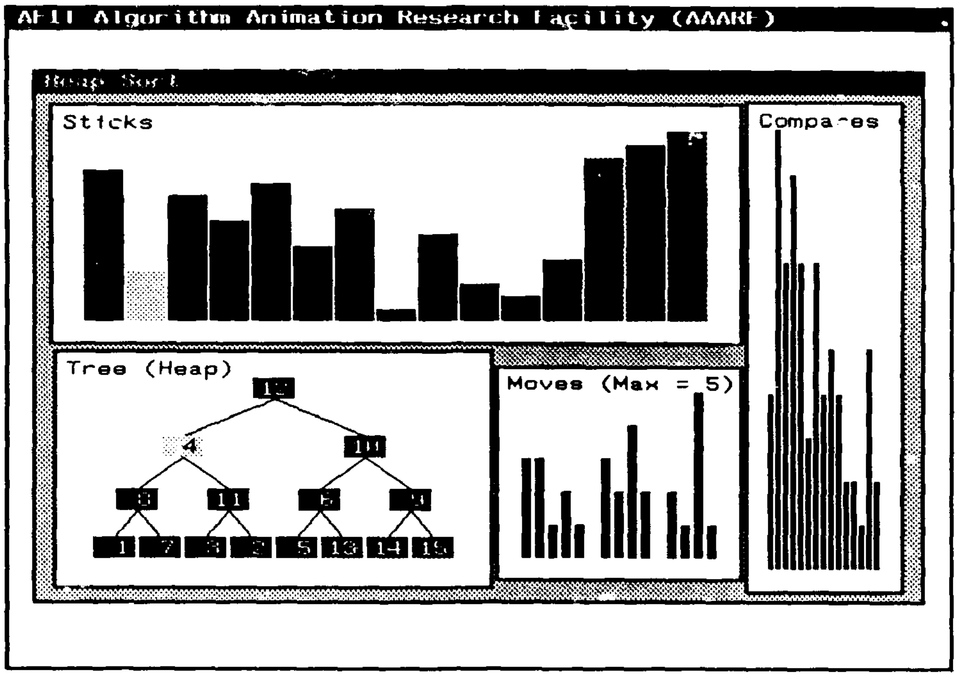


Figure 5.2. Four Views of a Heap Sort

5.5 *Recommendations for Further Research*

Based on the results of this study and the observations made during it, this section presents some recommendations for further research and some suggestions for enhancing AAARF.

The Value of Algorithm Animation. Before too much time is spent developing more algorithm animations, a study should be conducted to determine if there is sufficient cause to do so. Can end-users learn more or faster with algorithm animation than with conventional teaching methods? Is algorithm animation useful for debugging?

Remotely-Hosted Algorithm Components. The animation of algorithms executing on remote hosts is an interesting area of research. These animations could reveal not only aspects of the algorithm, but features of the host architecture as well. For example, a variety of interconnection networks could be compared for sorting arrays of numbers, performing matrix operations, or numerical integration on a parallel computer system.

Unimplemented Features. The remaining requirements of the enumerated requirements specification should be implemented (see Section 5 2). In particular, the library of animation primitives greatly increase the client-programmers ability to develop new animations.

Enhancements. Several enhancements could be made to increase the usefulness and improve the performance of AAARF:

Extended Recorder Capability. Every algorithm should be recorded as it is being animated. This would allow the user to hit reverse at any time to review a particularly interesting portion of an animation. While the software "tape" is playing, the background process waits with the next "live" interesting event. The transition from "tape" to "live" should be not be apparent to the end-user. The recorder panel could be eliminated and incorporated into the master control panel.

Recording Control Information. The animation recordings should also include control information — speed changes, starts, stops, and break-points. This is similar to the "scripts" used by Marc Brown in BALSA [6:71]. It allows a user to create a "movie" which emphasizes particular sequences that the user feels are particularly interesting or important.

Timing of Animation Updates. Currently AAARF is at the mercy of the SunOS scheduler with respect to the sequential updating of the displays for multiple algorithms. This can produce misleading results. There should be some way of (1) controlling which algorithm is next to update its display, and (2) applying a display time weighting factor to algorithm events that is proportional to the time that the event would actually take in execution. For example, in the *sort* algorithm class, the IN_PLACE interesting event is a conceptual event that actually takes no time to execute, but it is given the same amount of display time as as the EXCHANGE interesting event which obviously takes significantly more time in execution.

Automatic Views of Code. Perhaps the best ways of presenting an algorithm is a combination of text and graphics. An automatic display of the algorithm text along with the animation might improve a users understanding of the algorithm and be useful for debugging. The currently executing instruction could be highlighted within the text display.

Portability Problems. With respect to portability, AAARF actually has two problems. The first has to do with the resource requirements of AAARF. It's possible that AAARF could be optimized so that fewer resources are required for its execution. Also, a better estimate of the minimum system configuration should be developed. The second problem involves running AAARF on systems with monochrome monitors. AAARF should be modified to directly support monochrome systems by using a variety of pixel patterns to emulate color.

5.6 Summary

The system testing revealed that some requirements were not satisfied, but that, overall, the system performed according to design. An evaluation of AAARF showed that the goals of "developing a methodology for creating algorithm animations and developing an environment for controlling, displaying, and interacting with the animations" were satisfied. Conclusions of the study were presented and directions for future research were recommended.

Bibliography

1. Baecker, Ronald. *Sorting Out Sorting.* 16mm sound film, 25 min., Siggraph, 1981.
2. Bentley, Jon and Brian Kernighan. *A System for Algorithm Animation: Tutorial and User Manual.* AT&T Bell Laoratories, January 1987. Computing Science Technical Report No. 132.
3. Booch, Grady. *Software Components with Ada.* Menlo Park, CA: The Benjamin/Cummings Publishing, Inc., 1987.
4. Booch, Grady. *Software Engineering with Ada.* Menlo Park, CA: The Benjamin/Cummings Publishing, Inc., 1987.
5. Brown, Gretchen P., et al. "Program Visualization: Graphical Support for Software Development," *Computer*, *18*(8):27–35 (August 1985).
6. Brown, Marc H. *Algorithm Animation.* Cambridge, Massachusetts: The MIT Press, 1987.
7. Brown, Marc H. "Exploring Algorithms Using BALSA-II," *Computer*, *21*(5):14–36 (May 1988).
8. Brown, Marc H. and Robert Sedgewick. "A System for Algorithm Animation," *Computer Graphics*, *18*(3):177–186 (July 1984).
9. Brown, Marc H. and Robert Sedgewick. "Techniques for Algorithm Animation," *Software*, pages 28–39 (January 1985).
10. Chandry, K. Mani and Jayadev Misra. *Parallel Programmer Design.* Reading, Massachusetts: Addison-Wesley Publishing Company, 1988.
11. Dijkstra, Edsger W. "On the Cruelty of Really Teaching Computing Science," *SIGCSE Bulletin*, *21*(1):xxv-xxxix (February 1989).
12. Glinert, Ephraim, et al., "Introduction to Visual Programming Environments." ACM SIGGRAPH/89 16th Annual Conference on Computer Graphics and Interactive Techniques. Class notes.
13. Hartrum, Thomas C. *Readings II — Requirements Analysis.* EENG 593 Class notes, Draft #2, Air Force Institute of Technology, January 1988.
14. Hartrum, Thomas C. *System Development Documentation Guidelines and Standards.* Draft #4, Air Force Institute of Technology, January 1989.
15. Hartrum, Thomas C., et al. "Evaluating User Satisfaction of an Interactive Computer Program." In *Proceedings of IEEE 1989 National Aerospace and Electronics Conference (NAECON 89)*, pages 508–514, May 1989.

16. Horowitz, Ellis and Sartaj Sahni. *Data Structures*. Rockville, Maryland: Computer Science Press, 1983.

17. London, Ralph L. and Robert A. Duisberg. "Animating Programs Using SmallLalk," *Computer*, pages 61–71 (August 1985).

18. Moriconi, Mark and Dwight F. Hare. "Visualizing Program Design Through PegaSys," *Computer*, *18*(8):72–85 (August 1985).

19. Myers, Brad A. "Incense: A system For Displaying Data Structures," *Computer Graphics*, *17*(3):115–125 (July 1983).

20. Myers, Brad A. *The State of the Art in Visual Programming and Program Visualization*. Technical Report CMU-CS-88-114, Carnegie Mellon University Technical Report, February 1988.

21. Pressman, Roger S. *Software Engineering: A Practitioner's Approach*. New York: McGraw-Hill Book Co., 1982.

22. Scanlan, David. "Structured Flowcharts vs. Pseudocode: The Preference for learning Algorithms With a Graphical Method," *Engineering Education*, pages 173–177 (December 1988).

23. Stasko, John T. "Simplifying Algorithm Animation Design with TANGO." Georgia Institute of Technology, June 1989.

24. Sun Microsystems. *Getting Started with SunOS: Beginner's Guide*, 1988. SunOS Technical Documentation.

25. Sun Microsystems. *Network Programming*, 1988. SunOS Technical Documentation.

26. Sun Microsystems. *Pixrect Reference Guide*, 1988. SunOS Technical Documentation.

27. Sun Microsystems. *Programming Utilities & Libraries*, 1988. SunOS Technical Documentation.

28. Sun Microsystems. *SunView1 Programmer's Guide*, 1988. SunOS Technical Documentation.

Appendix A. *Detailed Design*

This appendix contains the detailed design structure charts for the AAARF implementation. The detailed design is concerned with developing an algorithmic abstraction for coordinating the activities of the preliminary design objects developed in Chapter III. The objects and operations are used to develop a high-level procedural description of the software. The following structure chart conventions are used:

- The symbol, SV, in the lower right corner of a design module indicates that the function is provided by SunView.
- The symbol, CP, in the lower right corner of a design module indicates that the function is required from the client-programmer.

There is a close, but not necessarily a one-to-one, correspondence between the design modules and the program modules. The *AAARF Programmer's Manual* provides more information regarding the actual program structure.

List of Figures

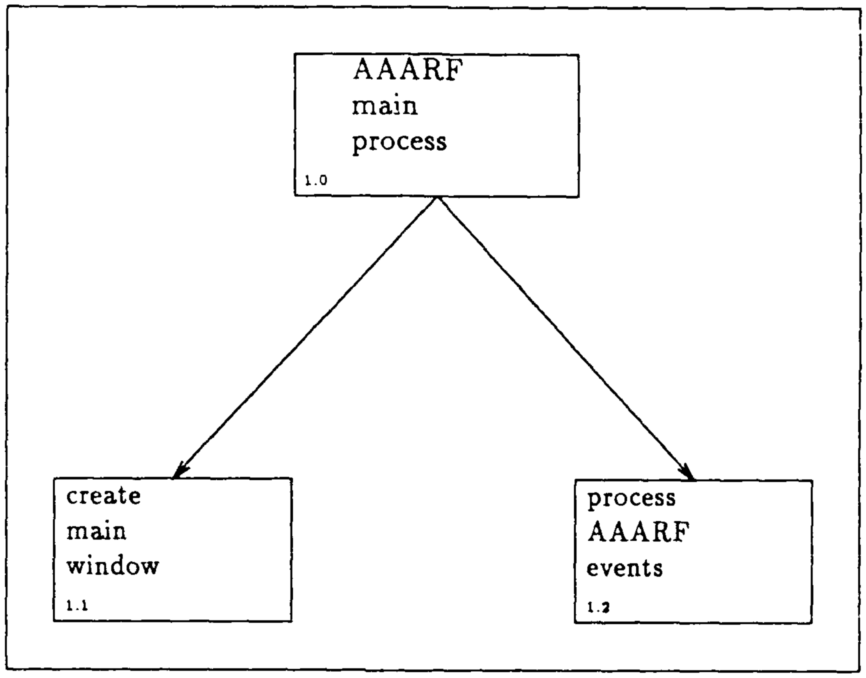


Figure 1: Design Module 1.0 AAARF Main Process

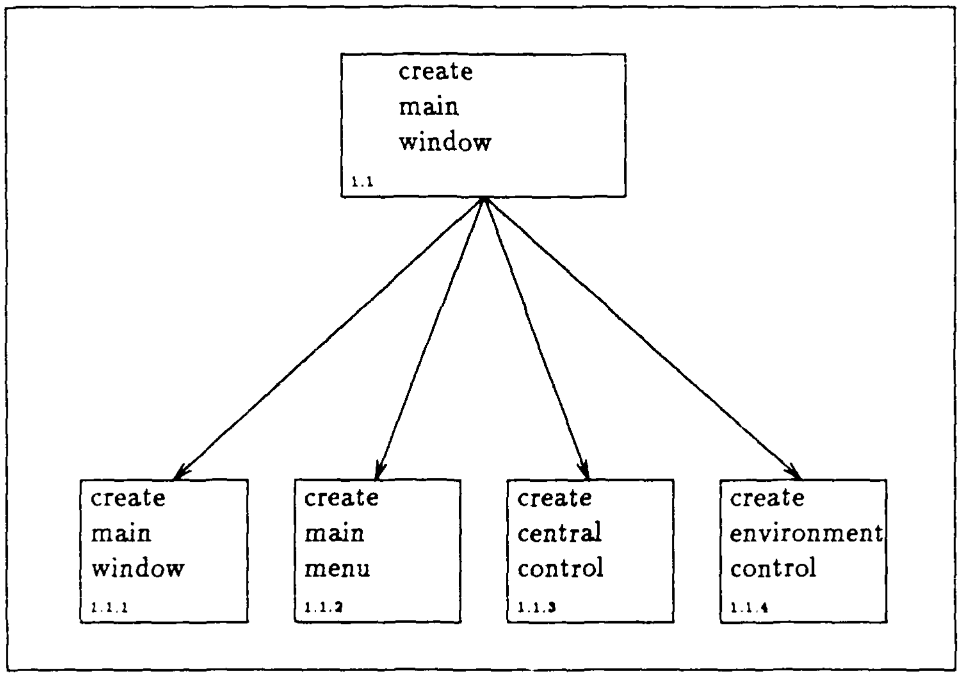


Figure 2: Design Module 1.1 Create Main Window

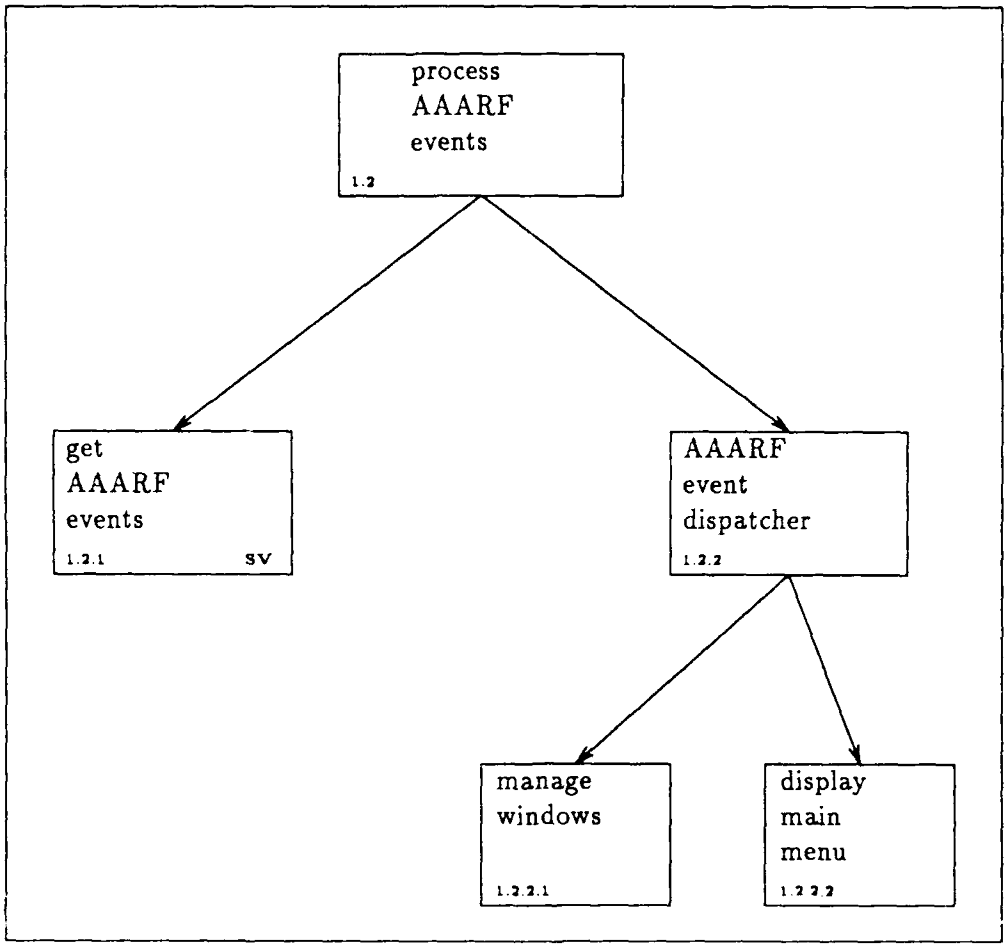


Figure 3: Design Module 1.2 Process AAARF Events

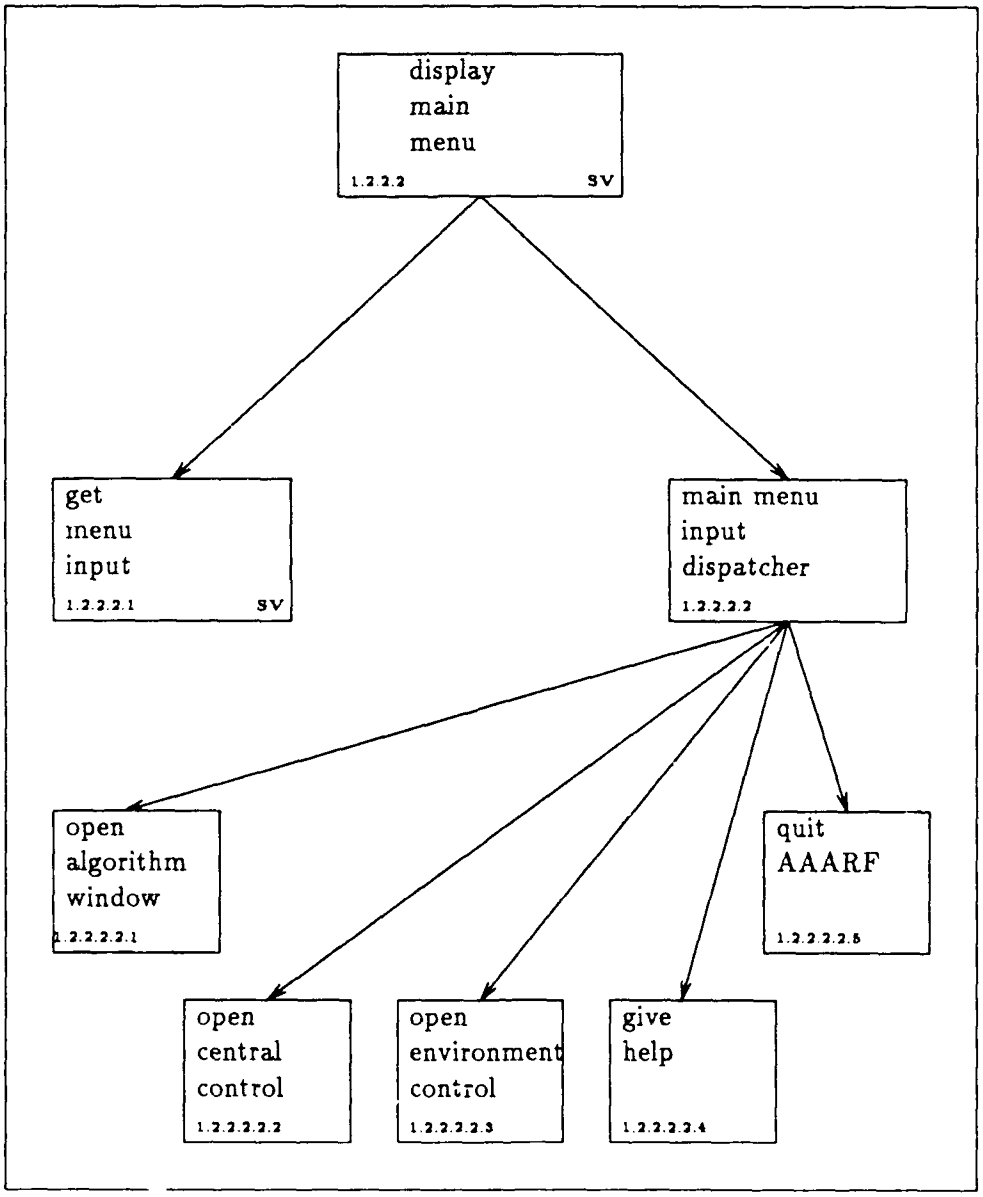


Figure 4: Design Module 1.2.2.2 Display Main Menu

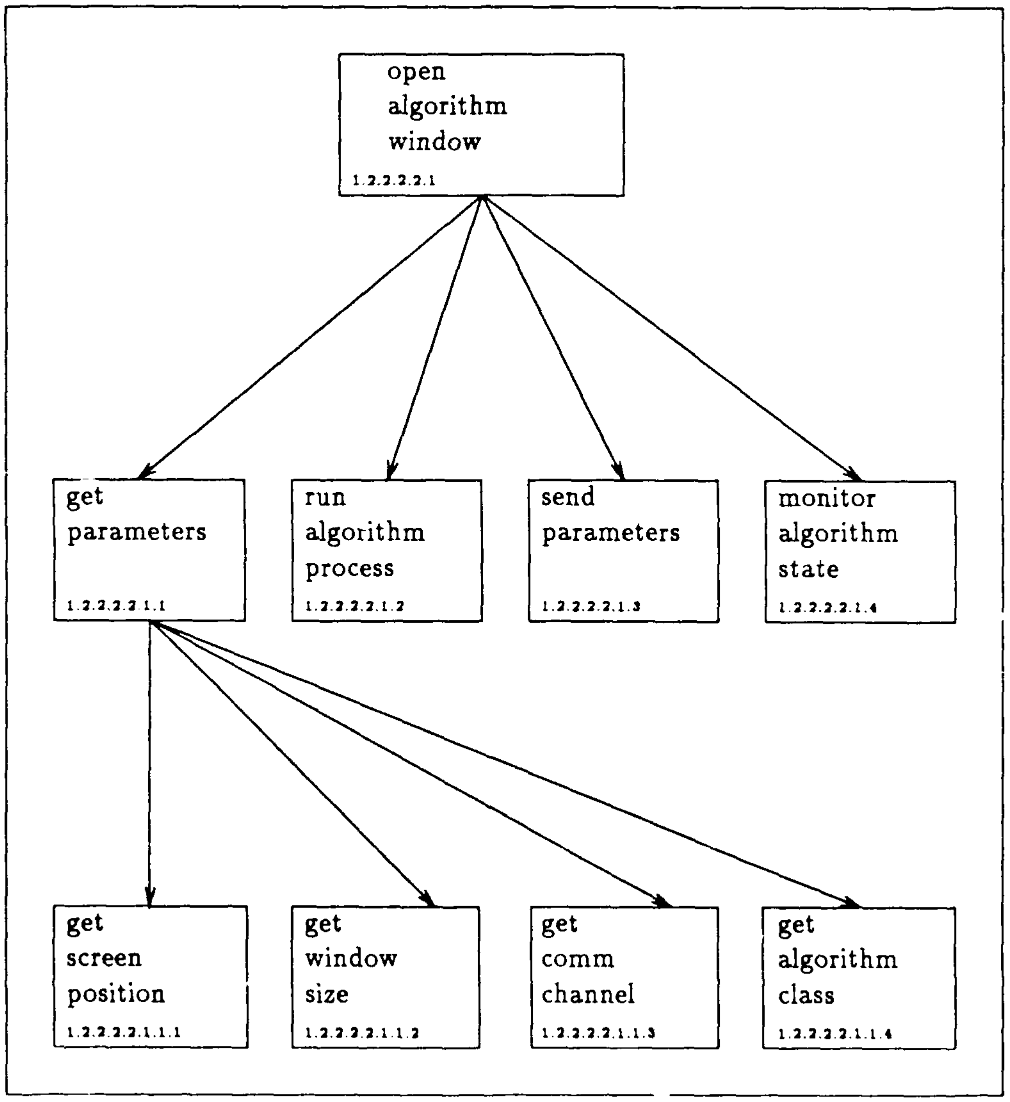


Figure 5: Design Module 1.2.2.2.2.1 Open Algorithm Window

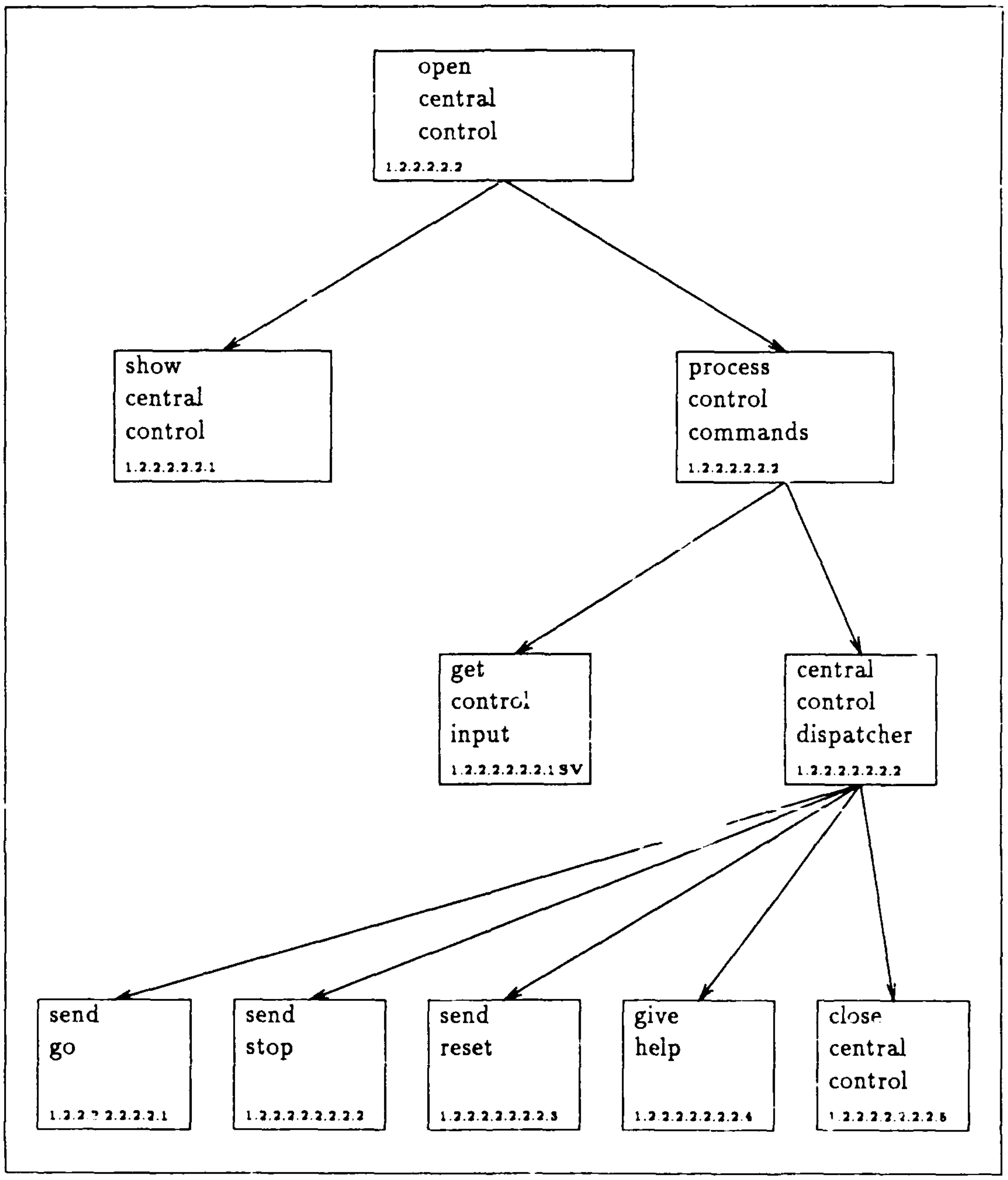

open central control 1.2.2.2.2.2
show central control 1.2.2.2.2.2.1
process control commands 1.2.2.2.2.2.2
get control input 1.2.2.2.2.2.2.1 SV
central control dispatcher 1.2.2.2.2.2.2.2
send go 1.2.2.2.2.2.2.2.1
send stop 1.2.2.2.2.2.2.2.2
send reset 1.2.2.2.2.2.2.2.3
give help 1.2.2.2.2.2.2.2.4
close central control 1.2.2.2.2.2.2.2.5

Figure 6: Design Module 1.2.2.2.2.2 Open Central Control

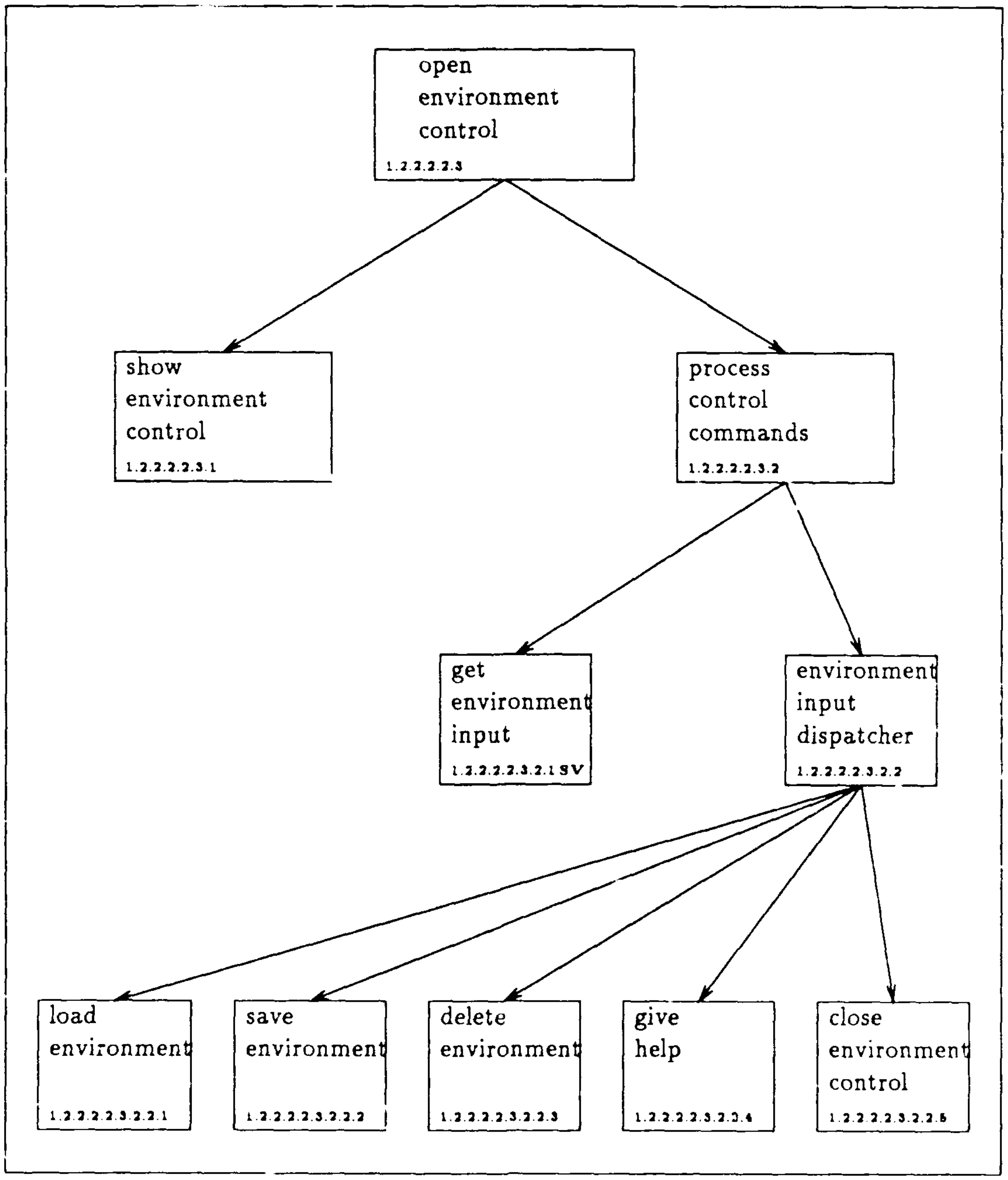


Figure 7: Design Module 1.2.2.2.2.3 Open Environment Control

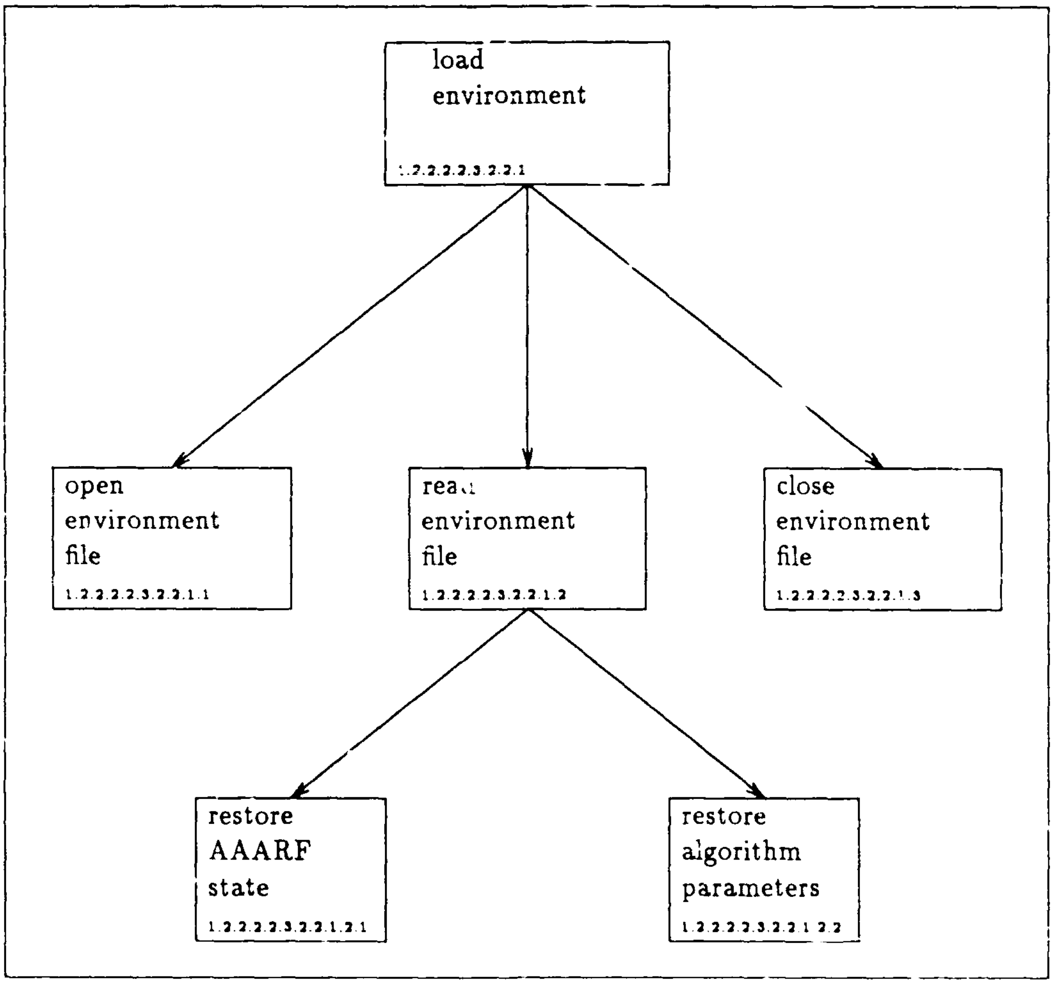


Figure 8: Design Module 1.2.2.2.2.3.2.2.1 Load Environment

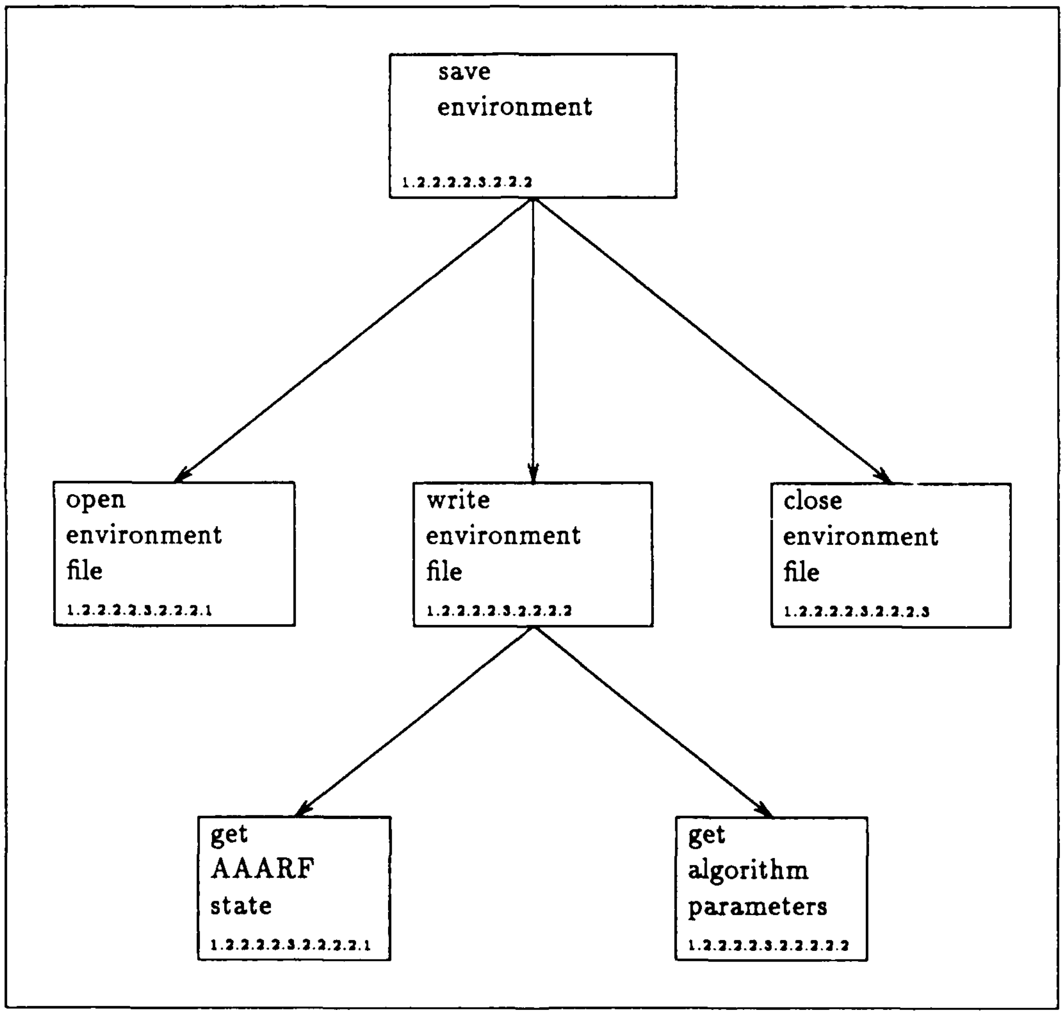


Figure 9: Design Module 1.2.2.2.2.3.2.2.2 Save Environment

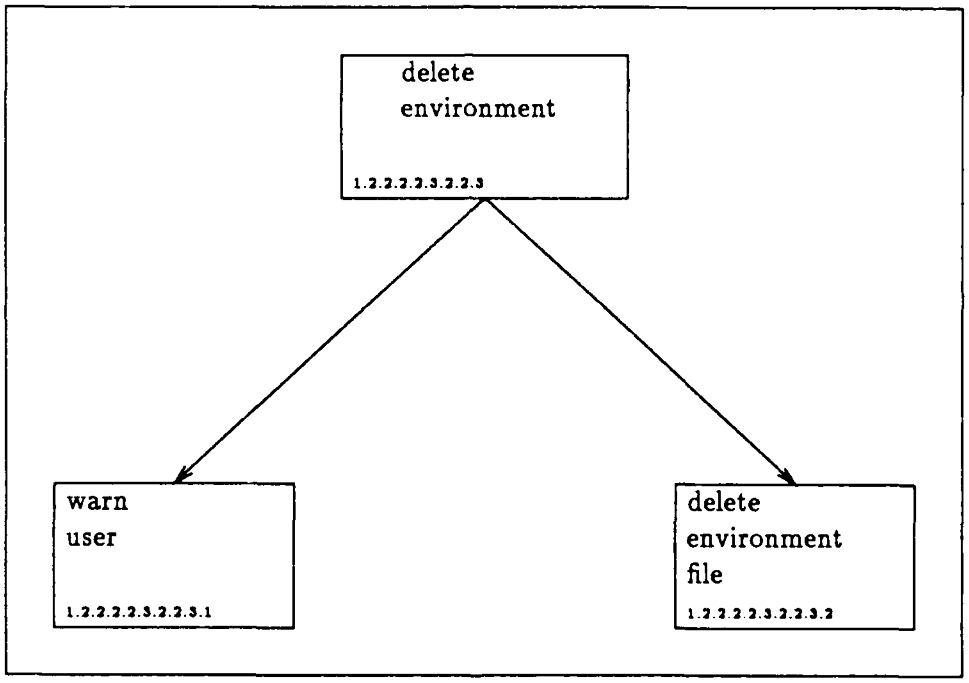


Figure 10: Design Module 1.2.2.2.2.3.2.2.3 Delete Environment

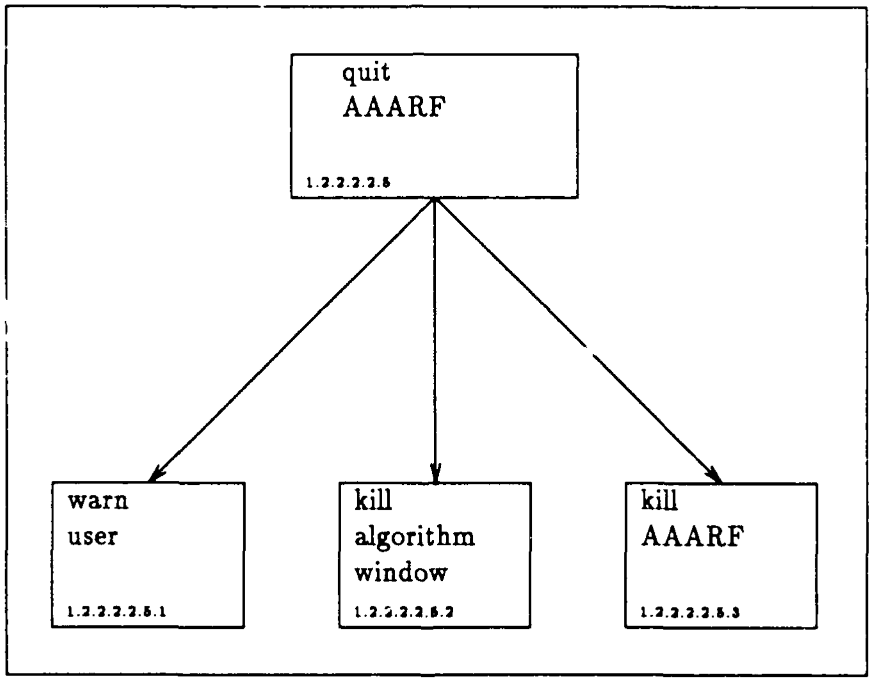


Figure 11: Design Module 1.2.2.2.2.5 Quit AAARF

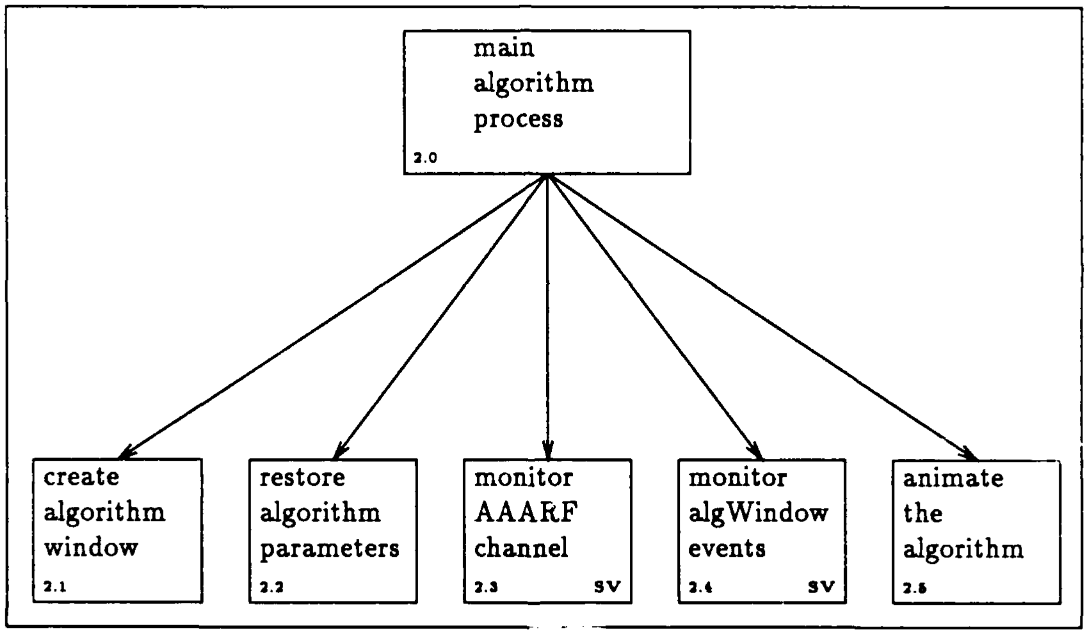


Figure 12: Design Module 2.. .ain Algorithm Process

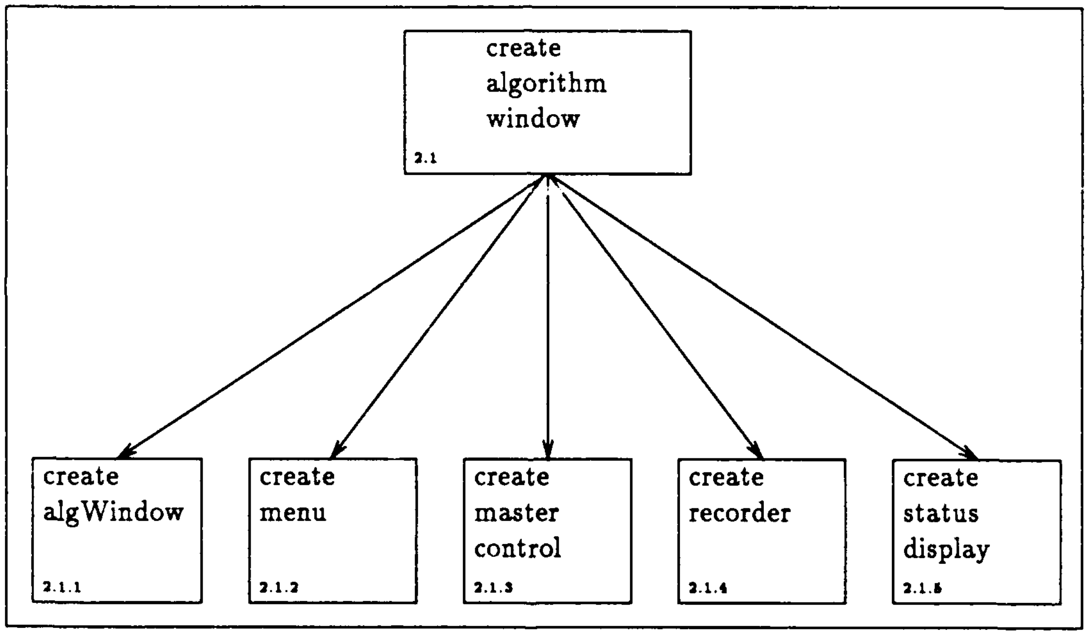


Figure 13: Design Module 2.1 Create Algorithm Window

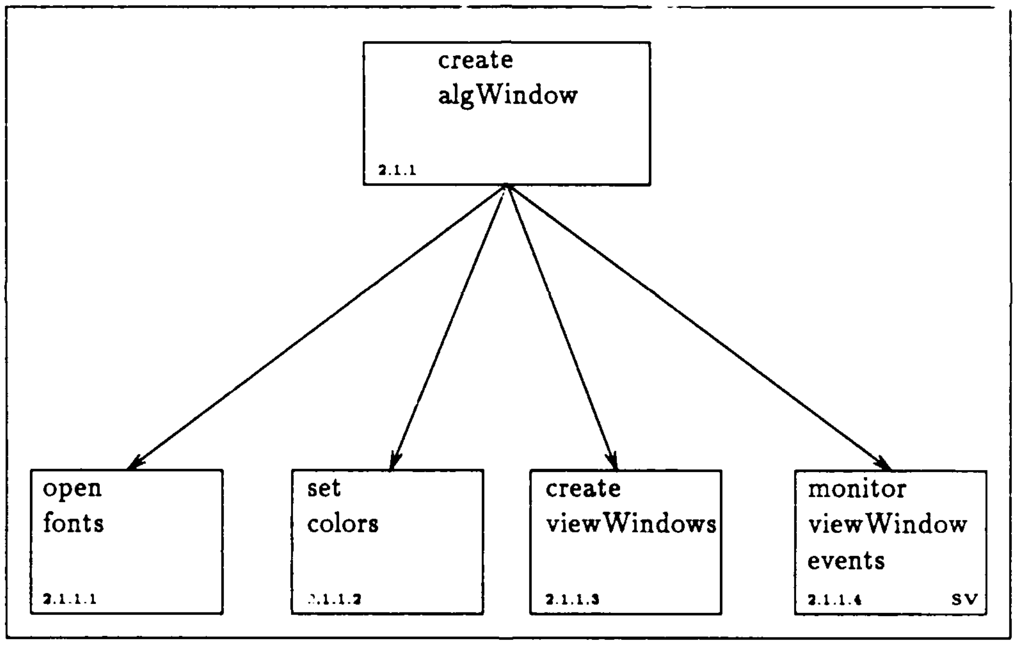


Figure 14: Design Module 2.1.1 Create AlgWindow

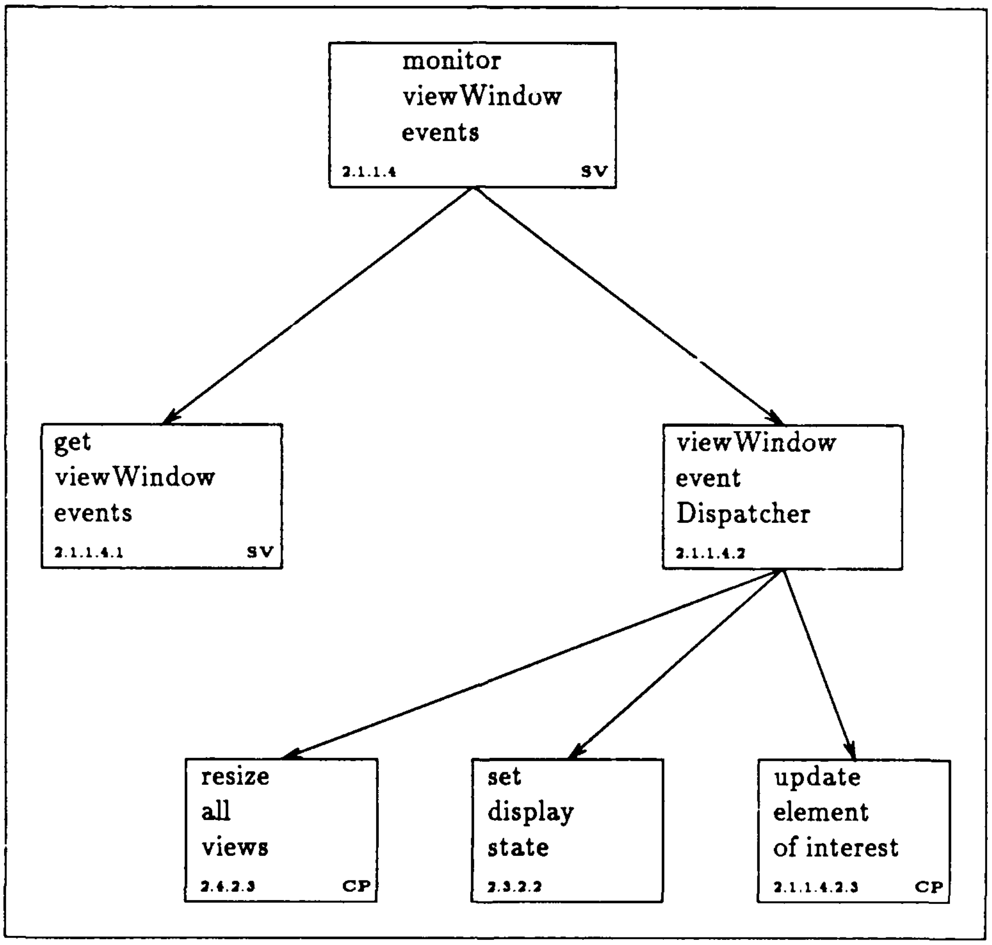


Figure 15: Design Module 2.1.1.4 Monitor View Window Events

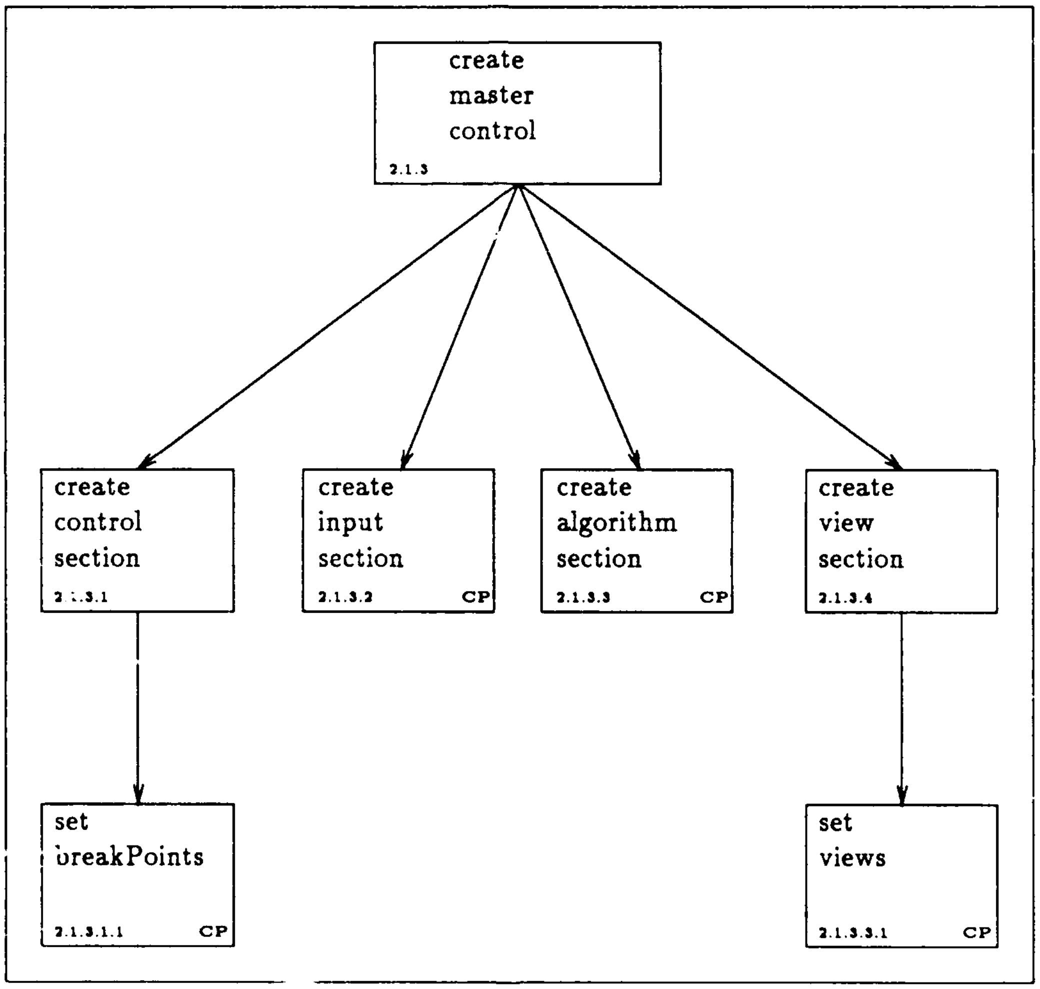


Figure 16: Design Module 2.1.3 Create Master Control

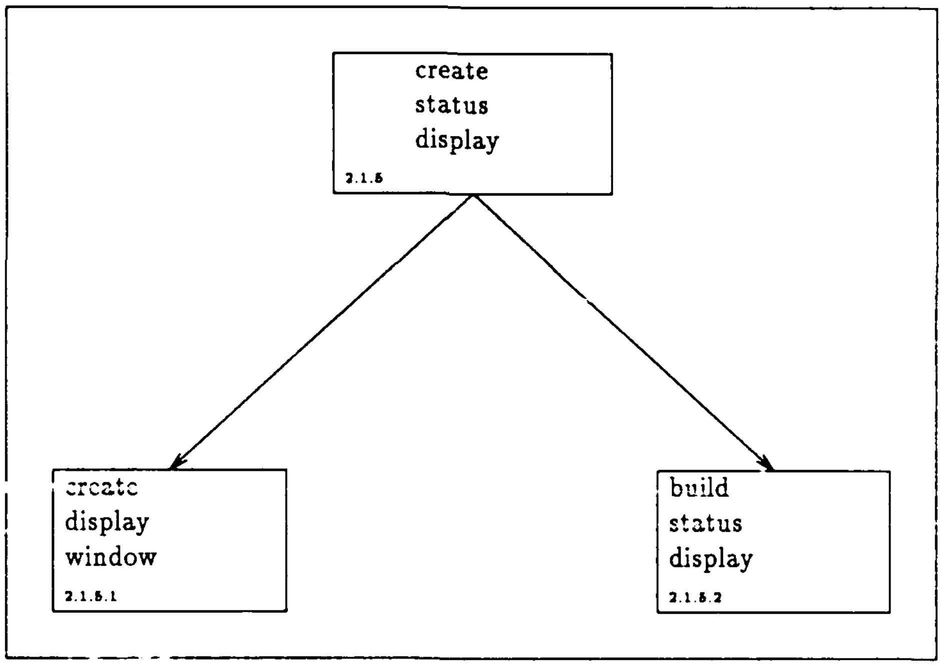


Figure 17: Design Module 2.1.5 Create Status Display

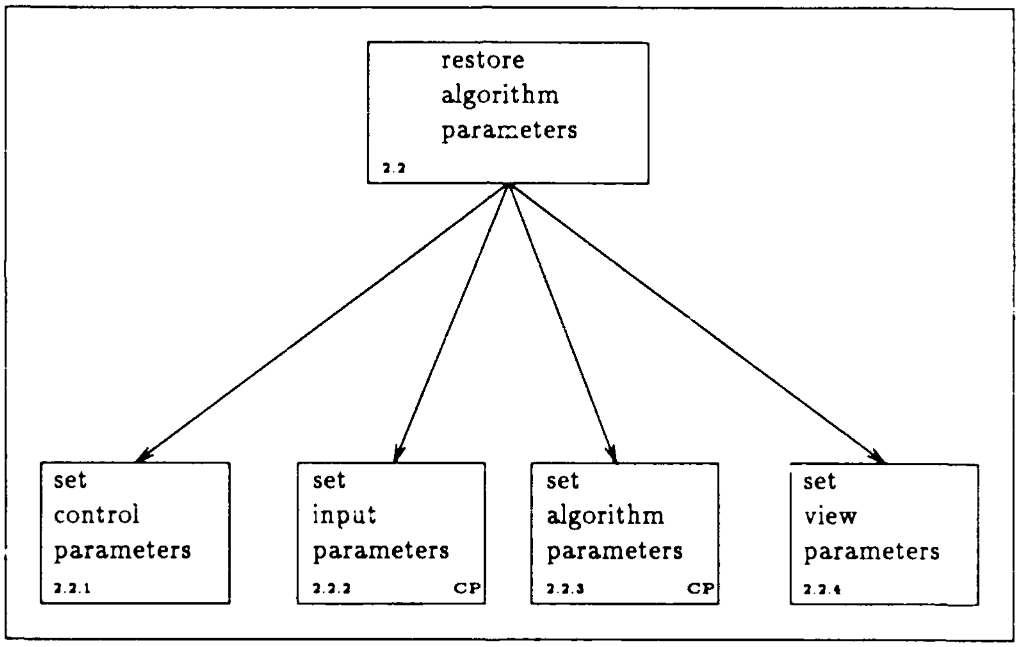


Figure 18: Design Module 2.2 Restore Algorithm Parameters

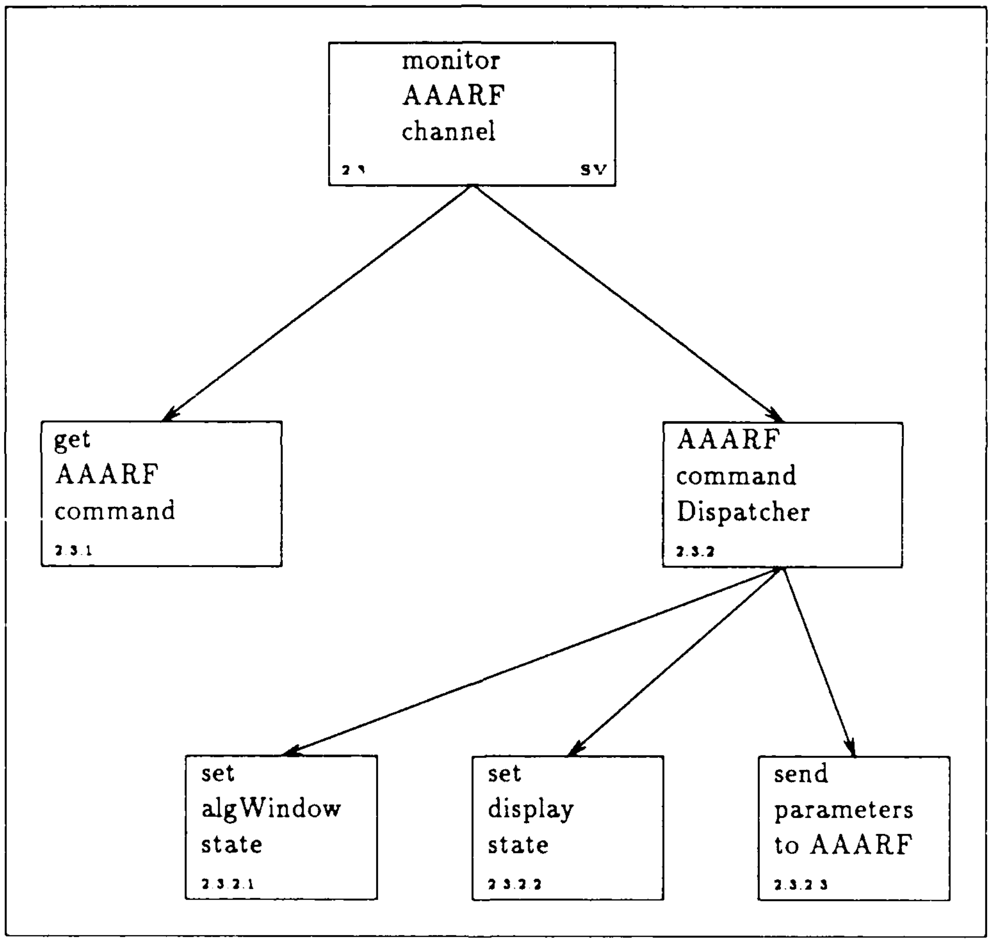


Figure 19: Design Module 2.3 Monitor AAARF Channel

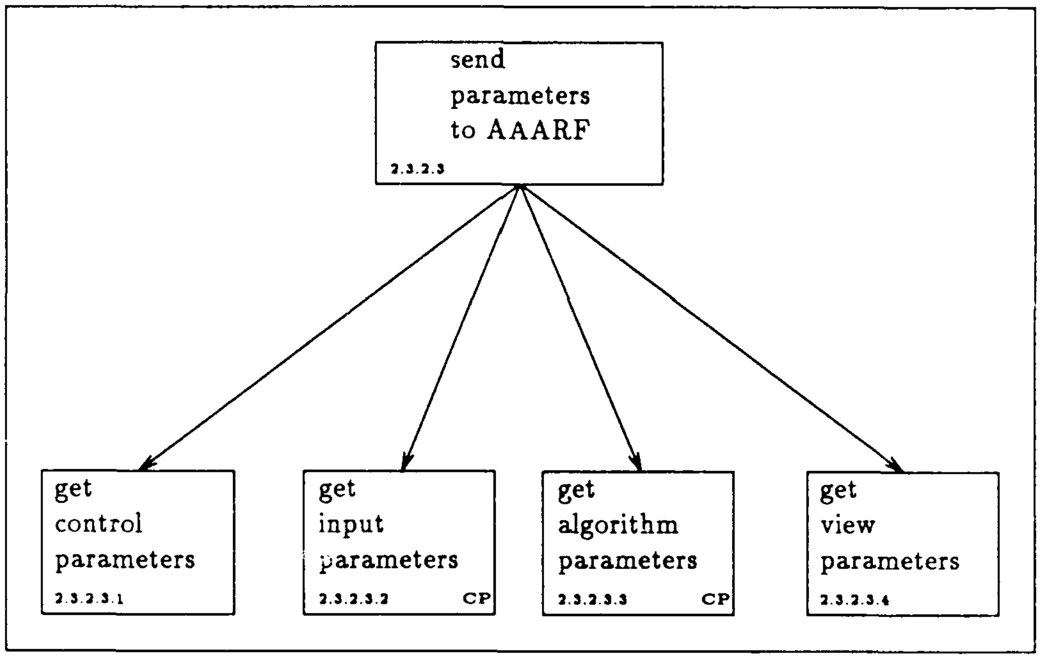


Figure 20: Design Module 2.3.2.3 Send Parameter to AAARF

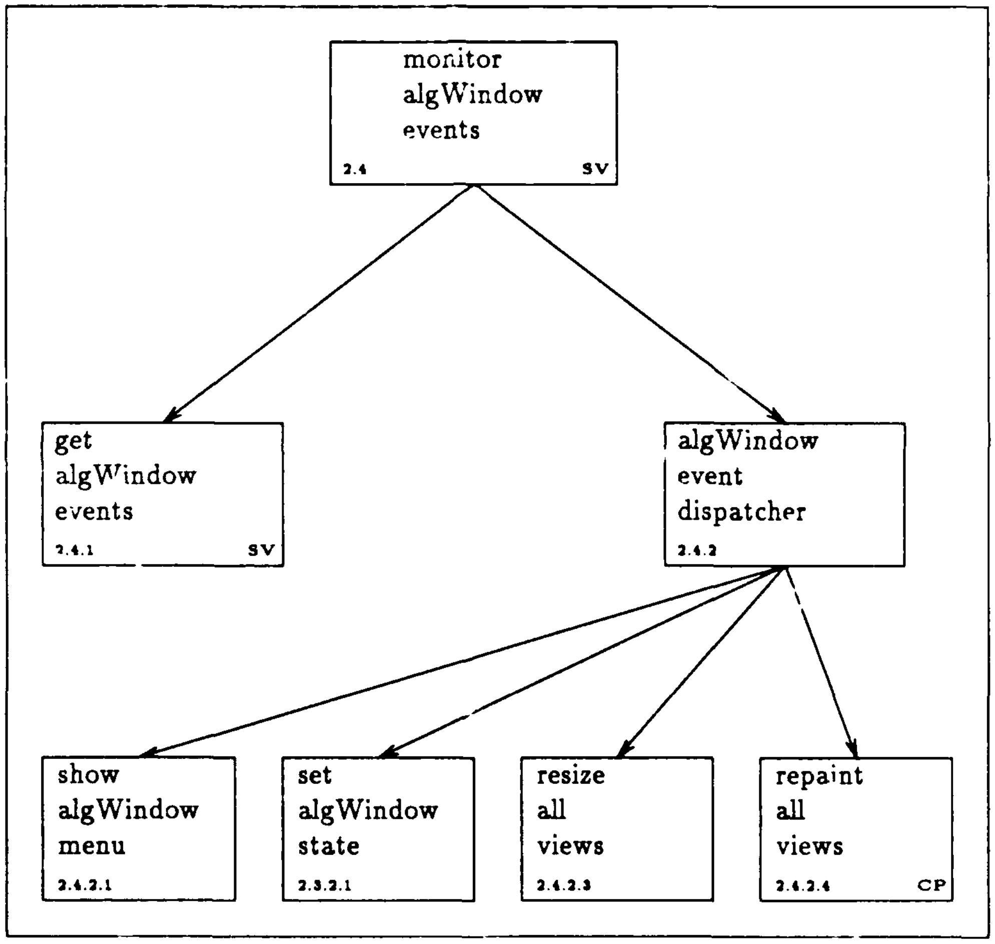


Figure 21: Design Module 2.4 Monitor AlgWindow Events

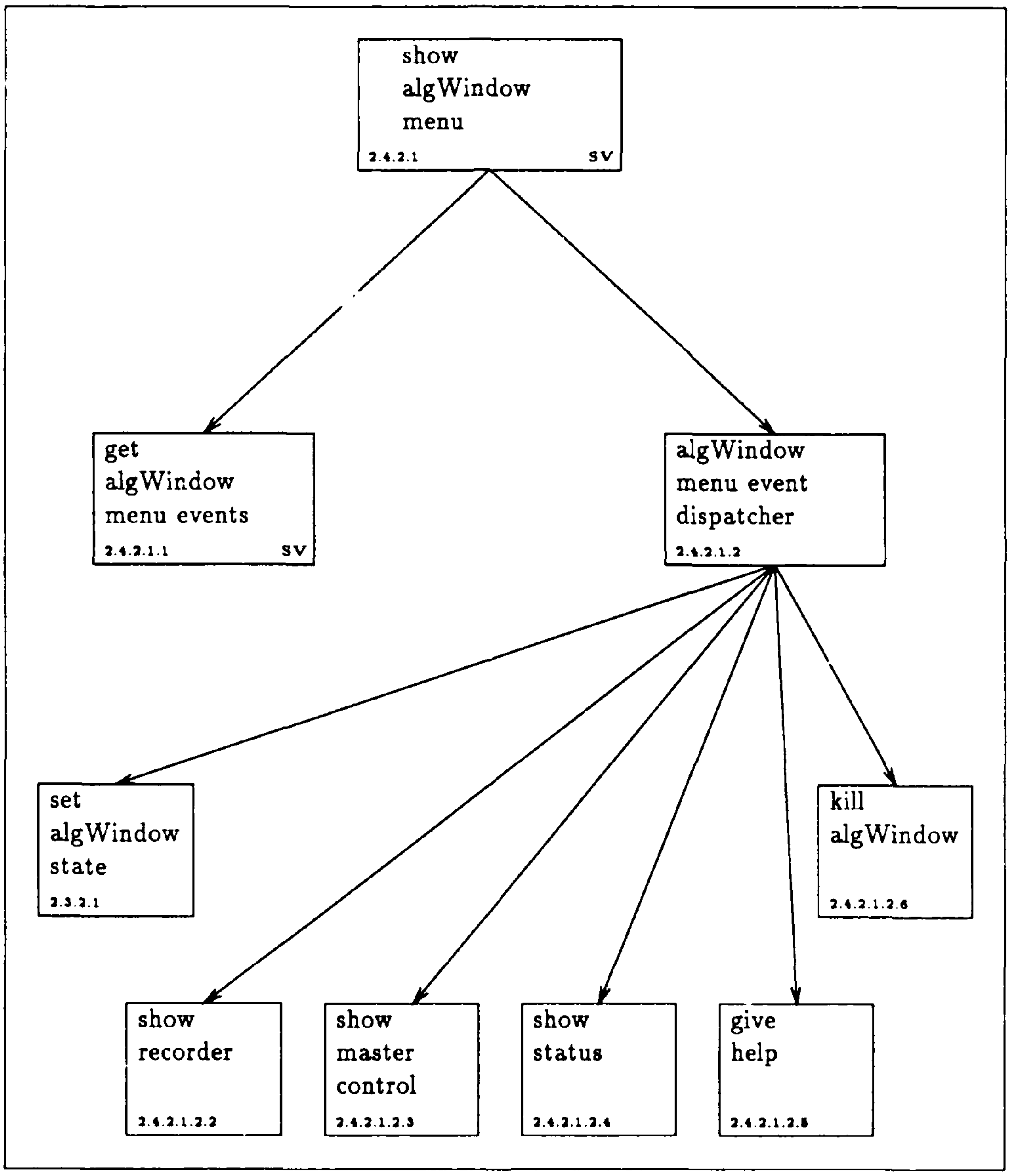


Figure 22: Design Module 2.4.2.1 Show AlgWindow Menu

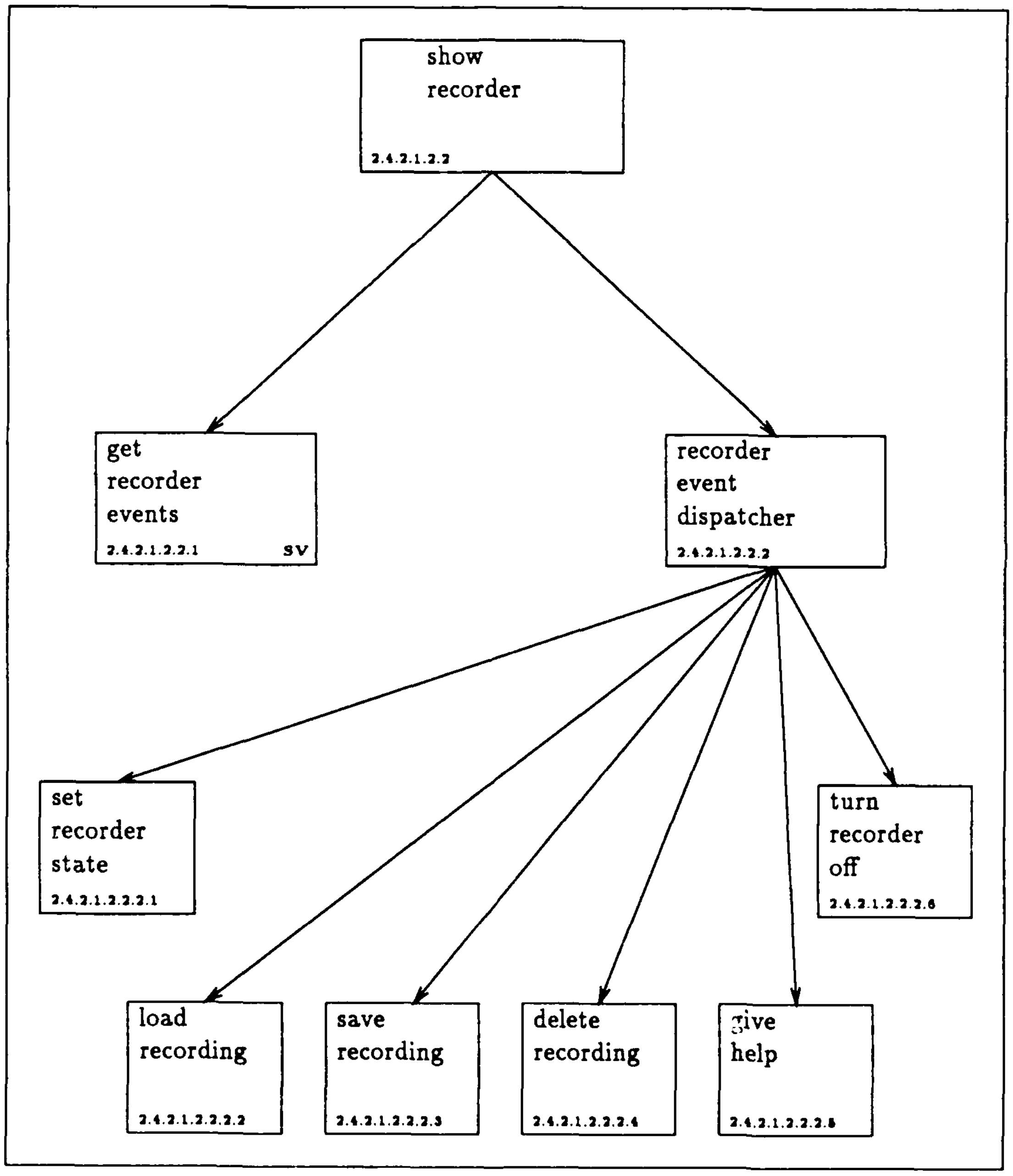


Figure 23: Design Module 2.4.2.1.2.2 Show Recorder

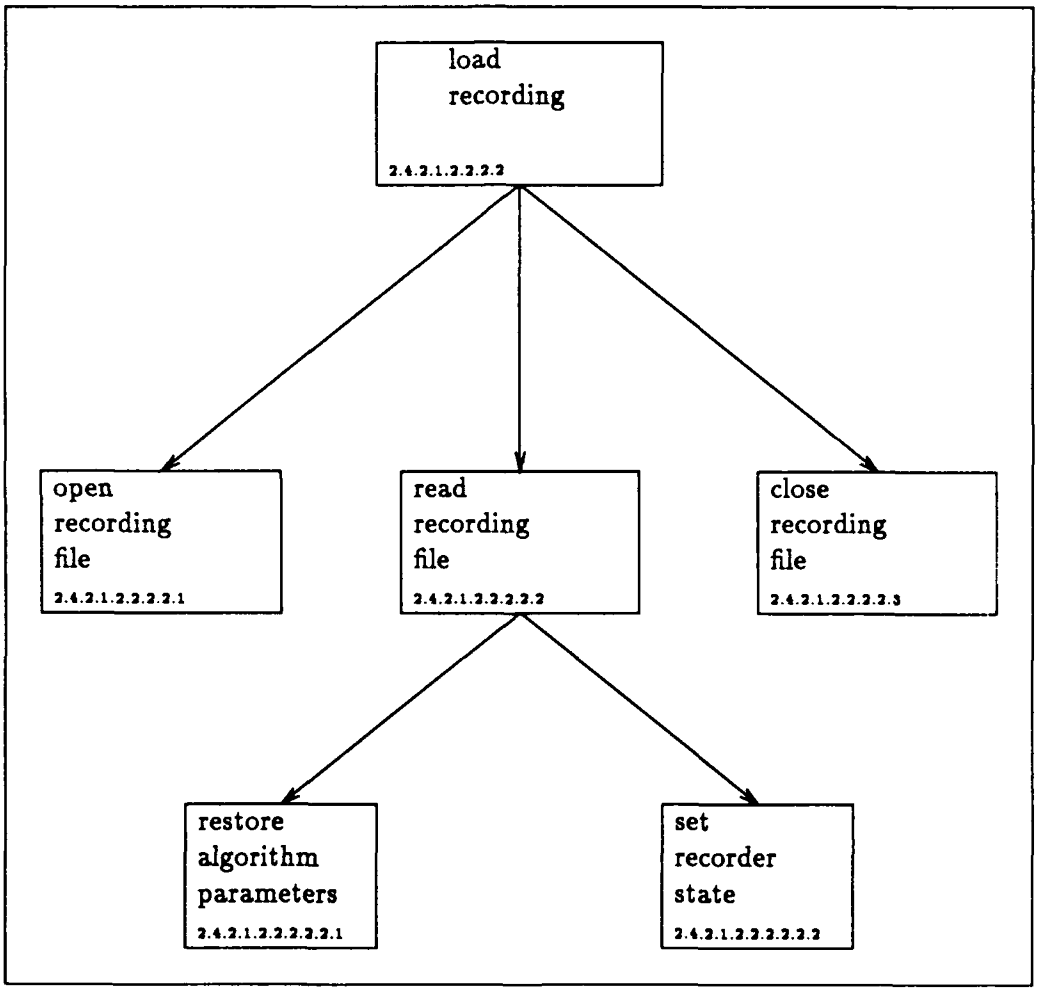


Figure 24: Design Module 2.4.2.1.2.2.2.2 Load Recording

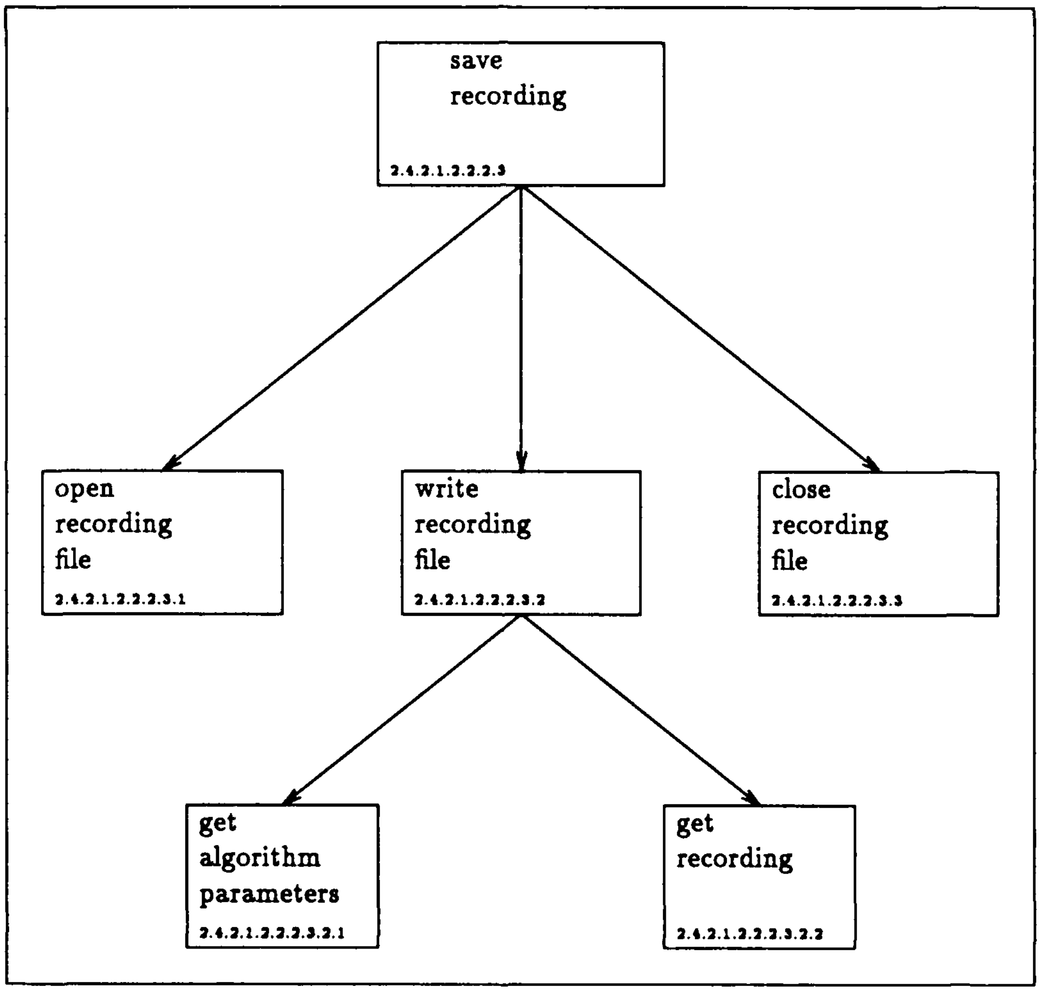


Figure 25: Design Module 2.4.2.1.2.2.2.3 Save Recording

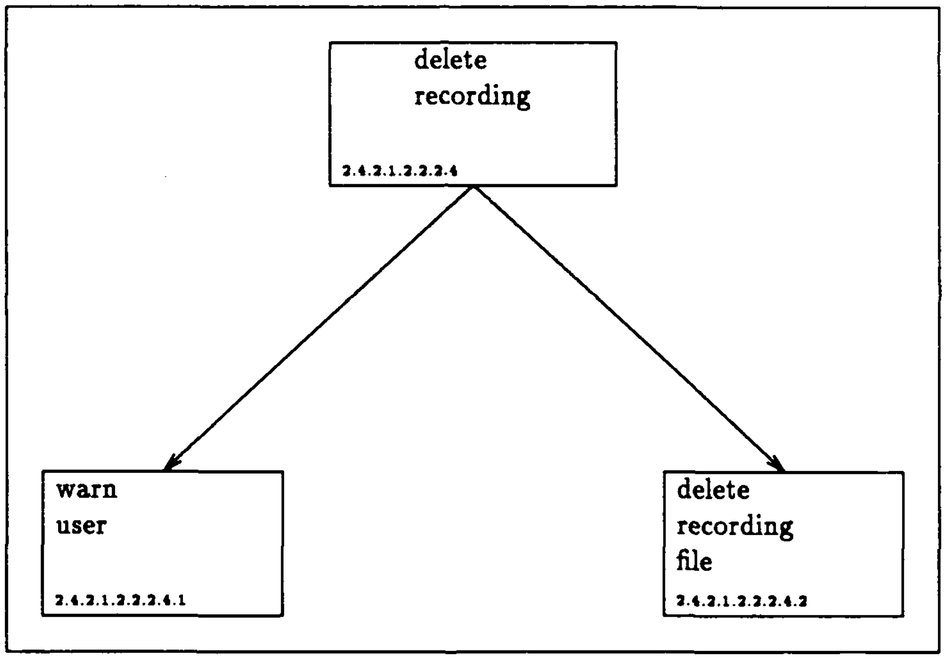


Figure 26: Design Module 2.4.2.1.2.2.2.4 Delete Recording

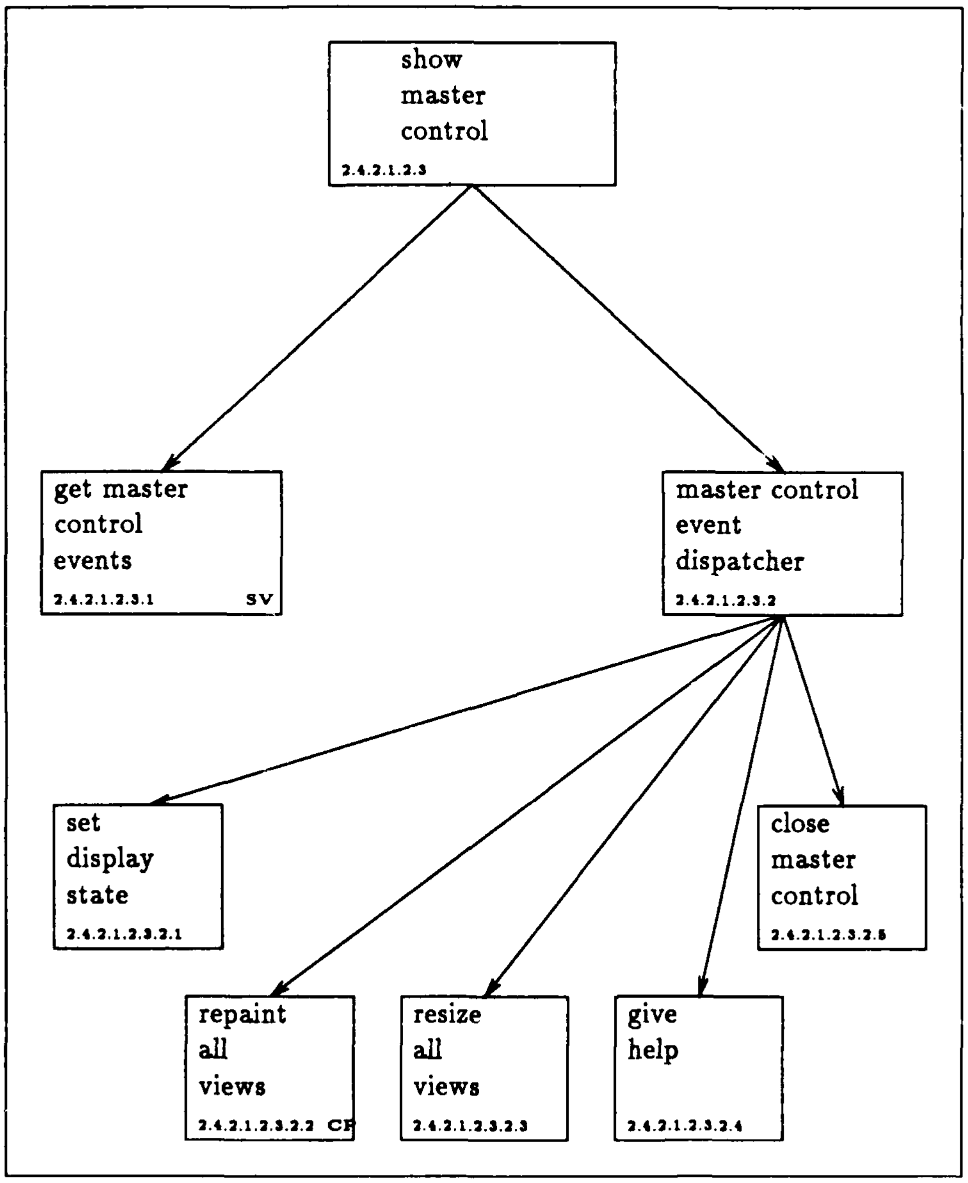


Figure 27: Design Module 2.4.2.1.2.3 Show Master Control

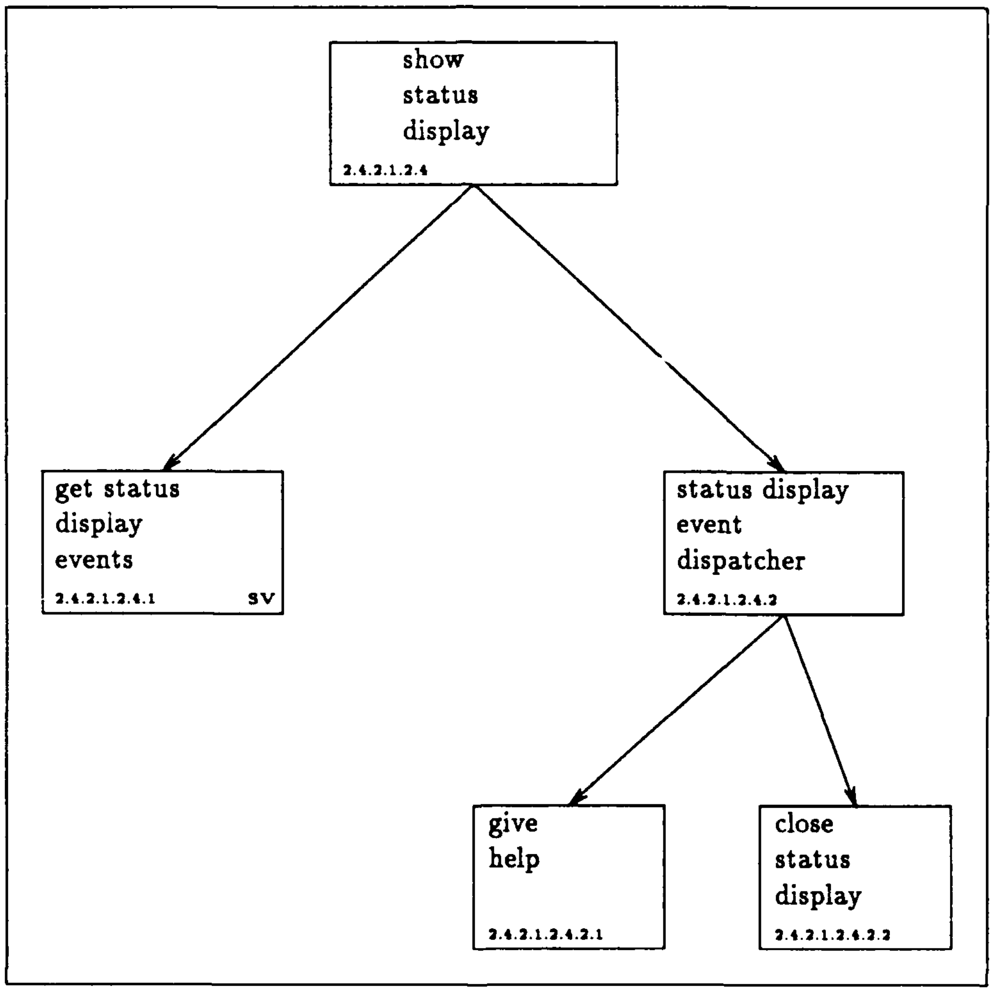


Figure 28: Design Module 2.4.2.1.2.4 Show Status

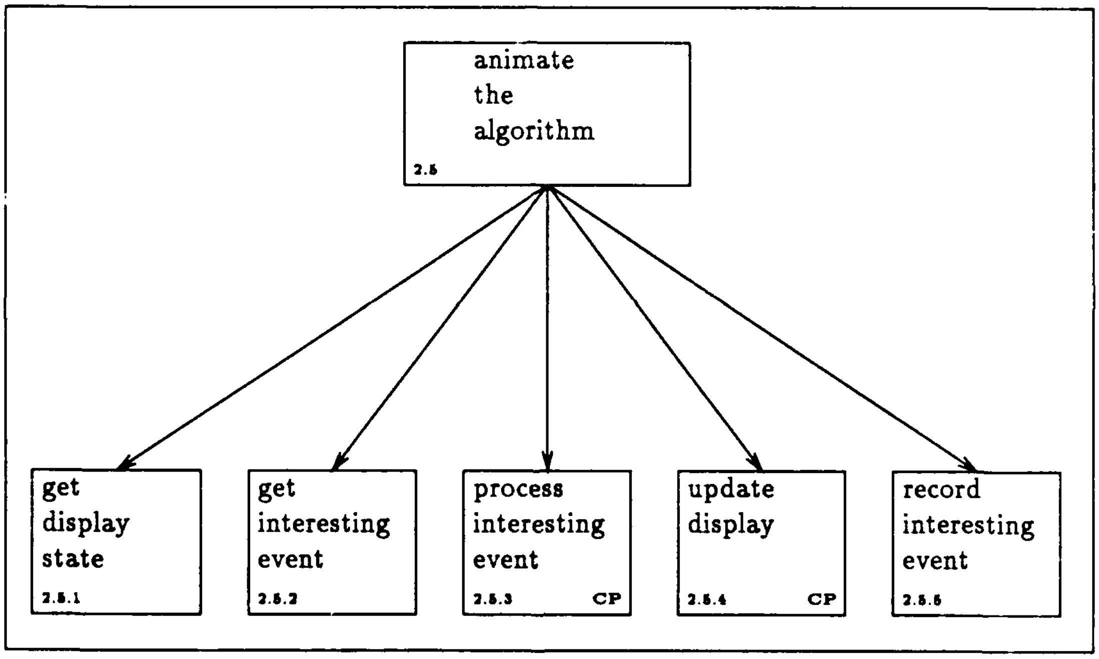


Figure 29: Design Module 2.5 Animate the Algorithm

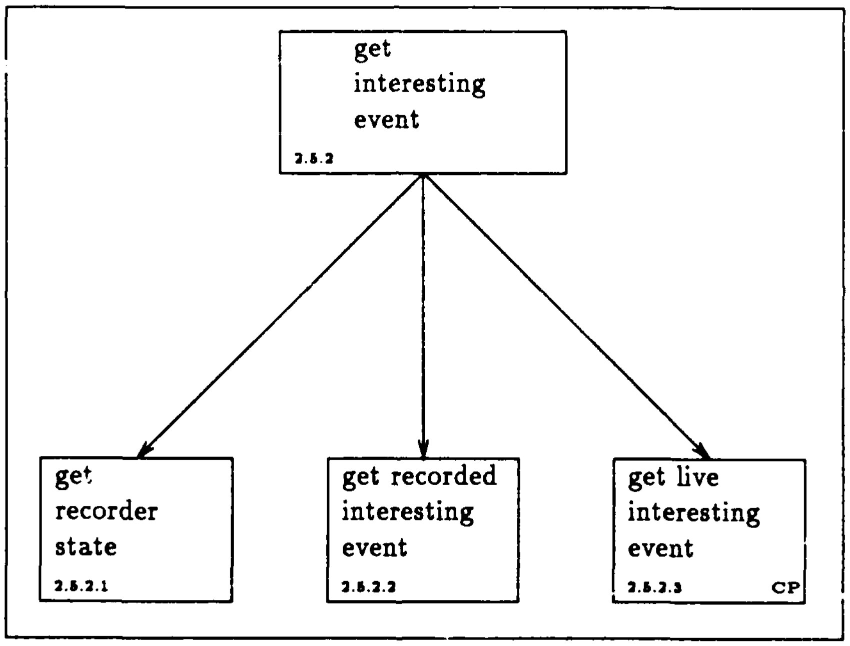


Figure 30: Design Module 2.5.2 Get Interesting Event

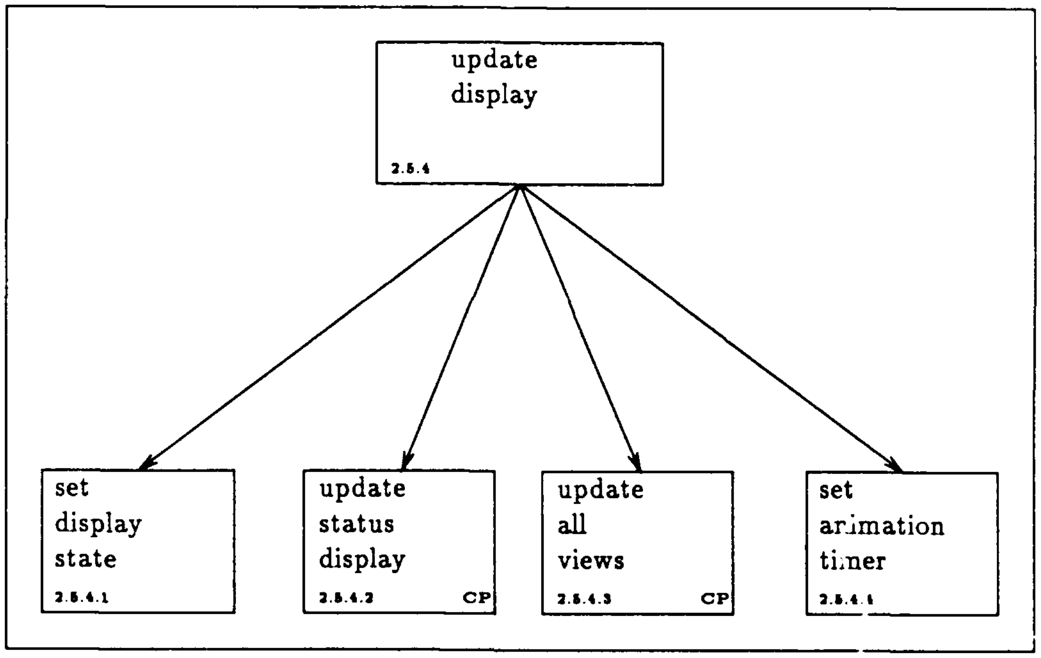


Figure 31: Design Module 2.5.4 Update Display

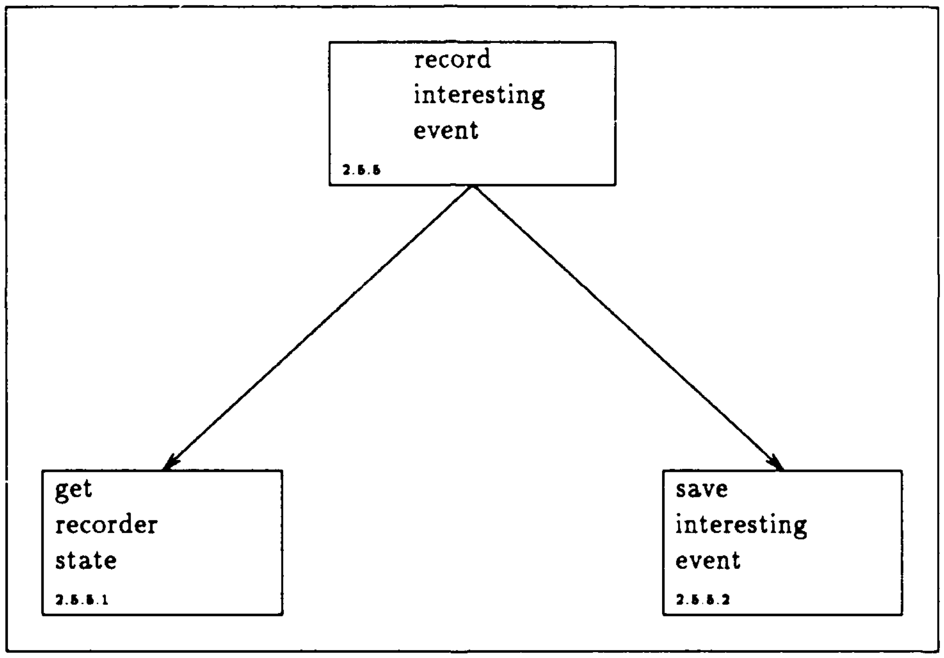


Figure 32: Design Module 2.5.5 Record Interesting Event

Appendix B. *AAARF User Evaluation Questionnaire*

This appendix contains an example of the user evaluation questionnaire for determining the effectiveness of the AAARF user interface and algorithm animations.

CAD-Tool Human-Computer Interface Evaluation[1]

Name (administrative use only): ______________________________.

Estimated time spent with tool/system [:]

Do not write in these spaces
Tool Evaluated:
Class:
Group:
Exper:
First:
ID#:

PLEASE READ BEFORE PROCEEDING:

The following questionnaire is designed to provide user feedback on the human-computer interface of the specified computer-aided design (CAD) tool. Through your responses, we hope to measure your degree of satisfaction with the tool, with primary emphasis on the "user-friendliness" of the human-computer interface.

The questionnaire consists of a set of 11 factors, plus an overall rating. We will determine your satisfaction with the tool based on your response to six adjective pairs used to describe each factor. Each adjective pair has a seven-interval range where you are to indicate your feelings with an "X". Responses placed in the center of the range will indicate that you have no strong feelings one way or the other, ***or that you cannot effectively evaluate that given factor.***

Evaluation begin time [:]

[1] 22 April 1988. U. S. Air Force Institute of Technology, AFIT/ENG(Hartrum)

1. *System Feedback or Content of the Information Displayed.* The extent to which the system kept you informed about what was going on in the program.

insufficient								sufficient
unclear								clear
useless								useful
bad								good
unsatisfactory								satisfactory

To me this factor is:

unimportant								important

Comments:

2. *Communication.* The methods used to communicate with the tool.

complex								simple
weak								powerful
bad								good
useless								useful
unsatisfactory								satisfactory

To me this factor is:

unimportant								important

Comments:

3. *Error Prevention* Your perception of how well the system prevented user induced errors.

bad								good
insufficient								sufficient
incomplete								complete
low								high
unsatisfactory								satisfactory

To me this factor is:

unimportant								important

Comments:

4. *Error Recovery.* The extent and ease with which the system allowed you to recover from user induced errors.

unforgiving								forgiving
incomplete								complete
complex								simple
slow								fast
unsatisfactory								satisfactory

To me this factor is:

unimportant								important

Comments:

5. *Documentation.* Your overall perception as to the usefulness of documentation.

useless								useful
incomplete								complete
hazy								clear
insufficient								sufficient
unsatisfactory								satisfactory

To me this factor is:

unimportant								important

Comments:

6. *Expectations.* Your perception as to the services provided by the system based on your expectations.

displeased								pleased
low								high
uncertain								definite
pessimistic								optimistic
unsatisfactory								satisfactory

To me this factor is:

unimportant								important

Comments:

7. *Confidence in the System.* Your feelings of assurance or certainty about the services provided by the system.

low	☐☐☐☐☐☐☐	high
weak	☐☐☐☐☐☐☐	strong
uncertain	☐☐☐☐☐☐☐	definite
bad	☐☐☐☐☐☐☐	good
unsatisfactory	☐☐☐☐☐☐☐	satisfactory

To me this factor is:

unimportant	☐☐☐☐☐☐☐	important

Comments:

8. *Ease of Learning.* Ease with which you were able to learn how to use the system to perform the intended task.

difficult	☐☐☐☐☐☐☐	easy
confusing	☐☐☐☐☐☐☐	clear
complex	☐☐☐☐☐☐☐	simple
slow	☐☐☐☐☐☐☐	fast
unsatisfactory	☐☐☐☐☐☐☐	satisfactory

To me this factor is:

unimportant	☐☐☐☐☐☐☐	important

Comments:

9. *Display of Information.* The manner in which both program control and data information were displayed on the screen.

confusing	☐☐☐☐☐☐☐	clear
cluttered	☐☐☐☐☐☐☐	well defined
incomplete	☐☐☐☐☐☐☐	complete
complex	☐☐☐☐☐☐☐	simple
unsatisfactory	☐☐☐☐☐☐☐	satisfactory

To me this factor is:

unimportant	☐☐☐☐☐☐☐	important

Comments:

10. *Feeling of Control.* Your ability to direct or control the activities performed by the tool.

low								high
insufficient								sufficient
vague								precise
weak								strong
unsatisfactory								satisfactory

To me this factor is:

unimportant								important

Comments:

11. ***Relevancy or System Usefulness.*** **Your perception of how useful the system is as an aid to a software developer.**

useless								useful
inadequate								adequate
hazy								clear
insufficient								sufficient
unsatisfactory								satisfactory

To me this factor is:

unimportant								important

Comments:

12. ***Overall Evaluation of the System.*** **Your overall satisfaction with the system.**

unsatisfied								satisfied

(cont'd)

Comments on the Overall System:

Evaluation end time [:]

Total time spent on evaluation [:]

Thank you for your help.

Appendix C. *The AAARF User's Manual*

The *AAARF User's Manual* details the operation of the AAARF program. It is intended to be a stand-alone document.

AAARF User's Manual

Keith Carson Fife
Captain, USAF

Department of Electrical and Computer Engineering
School of Engineering
Air Force Institute of Technology
Wright-Patterson Air Force Base, Ohio 45432

December, 1989

Contents

List of Figures

AAARF User's Manual

Captain Keith C. Fife

1 Introduction

The AFIT Algorithm Animation Research Facility (AAARF) is an interactive algorithm animation system. It provides a means for visualizing the execution of algorithms and their associated data structures. AAARF allows the user to select the type of algorithm, the input to the algorithm, and the views of the algorithm. Several control mechanisms are provided, including stop, go, reset, variable speed, single-step, and break-points. Other features of AAARF include:

- Multiple Algorithm Windows
- Simultaneous Control of Multiple Animations
- Animation Environment Save and Restore Capability
- Multiple View Windows within each Algorithm Window
- Animation Record and Playback
- Algorithm State Display and Interrogation Capability
- Master Control Panel for monitoring and modifying the input, algorithm, view, and control parameters to an algorithm animation.

AAARF runs on Sun3^{TM} and Sun4^{TM} workstations using the SunOS^{TM} (Sun Microsystem's version of the AT&T UNIX^{TM} operating system) and the SunView^{TM} window-based environment. AAARF is designed for use with color monitors. Monochrome monitors can be used, but the displays are not as informative.

AAARF has two types of users: *end-users* who view and interact with the algorithm animations, and *client-programmers* who develop and maintain the algorithms and animations. **This manual is primarily for end-users**; it describes how to use AAARF. Client-programmers should refer to the *AAARF Programmer's Manual* [2].

This manual introduces users to algorithm animation and AAARF. It explains how AAARF can be used to explore algorithms in ways not previously possible. No previous experience with algorithm animation is required; interested readers may refer to *The Graphical Representation of Algorithmic Processes*[3] for more detailed information. Though not necessary, some familiarity with SunOS and SunView is helpful. The following references are recommended:

- *Getting Started with SunOS: Beginner's Guide* [4]
- *Setting Up Your SunOS Environment: Beginner's Guide* [5]
- *The SunView1 Beginner's Guide* [6]

The next section introduces algorithm animation, describes the AAARF system architecture, and presents an overview of AAARF's use of SunView. Users should be thoroughly familiar with the concepts presented in this section before using AAARF. Section 3 explains how to start the AAARF program and describes the major components of the AAARF main screen. Section 4 discusses algorithm windows and explains how to start and control animations. Section 5 presents some exercises to test and develop an understanding of AAARF.

2 Overview

This section examines the general architecture of AAARF and introduces some SunView concepts necessary for understanding and effectively using AAARF. The definitions that follow are extracted from [3] and [1]; they are critical to an exact understanding of the concepts of this section:

Animation Components An algorithm animation consists of three components: an input generator, an algorithm, and one or more animation views (see Figure 1).

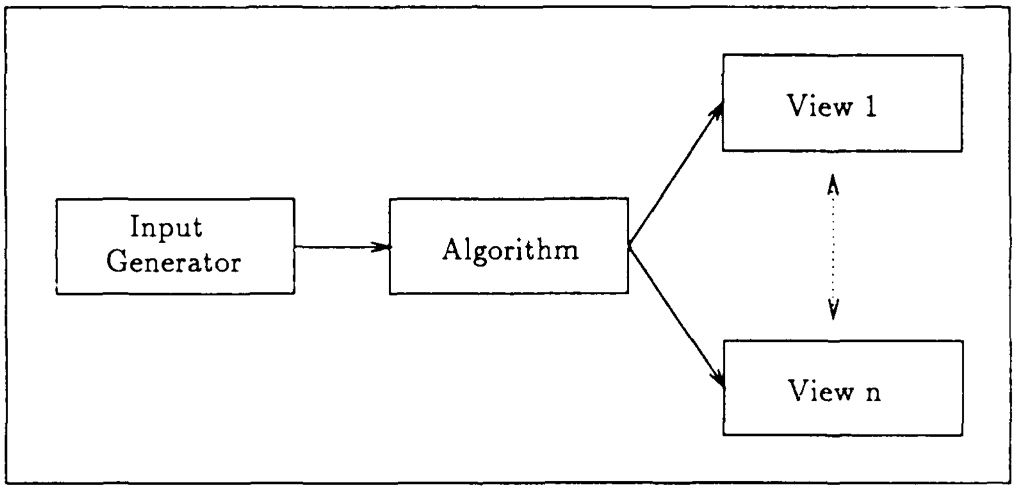


Figure 1: Algorithm Animation Components

Input Generator An input generator is a procedure which provides input to an algorithm; the input may be generated randomly, read from a file, or entered by the user.

Animation View Animation views are graphical representations of an algorithm's execution.

Interesting Event (IE) Animation views are driven by interesting events that occur during the execution of an algorithm. Interesting events are input events, output events, and state changes that an algorithm undergoes during its execution. The type, quantity, and sequence of IEs for a particular algorithm distinguish it from other algorithms.

Algorithm Classes Algorithms which operate on identical data structures and perform identical functions are from the same algorithm class. The *Array Sort* class includes *quick sort*, *heap sort*, *bubble sort*, and other in-place sort algorithms. Algorithms from the same class generally share input generators and views, although certain views and input generators may be ineffective with particular algorithms. For instance, the *tree* view is very meaningful for heap sort, but nearly useless for any other sort.

Parameterized Control User-selectable parameters are associated with each component. *Algorithm parameters* affect some aspect of how the algorithm executes. For example, with a quick sort algorithm, what partitioning strategy should be used; as the partitions get smaller, at what point should another type of sort be used; what other type of sort should be used. *Input Parameters* affect the input generator — what seed is used to generate a set of random numbers; how "sorted" is a set of unsorted numbers; what is the general form of a series of numbers. *View parameters* affect how the animation is displayed in the view window. For example, what shape is associated with the nodes in a graph; how should an arbitrary graph be positioned.

2.1 AAARF Architecture

AAARF uses three levels of execution linked via UNIX sockets to animate algorithms. Figure 2 shows the levels of execution.

AAARF Main Process.

AAARF's top-level process provides the main screen, a mechanism for saving and restoring animation environments, a mechanism for controlling multiple algorithm animations simultaneously, and a means for starting new algorithm animations. This level serves as a high-level manager of the animation environment.

- *In general, The AAARF main process is the only level of execution with which end-users need be concerned.*

Class-Specific Window-Based Process.

The second level of execution is the algorithm class-specific window-based process. The class-specific level of execution provides the animation views, an animation recorder, a status display for interrogating the state of the algorithm, and a master control panel for monitoring and modifying the parameters that control the animation.

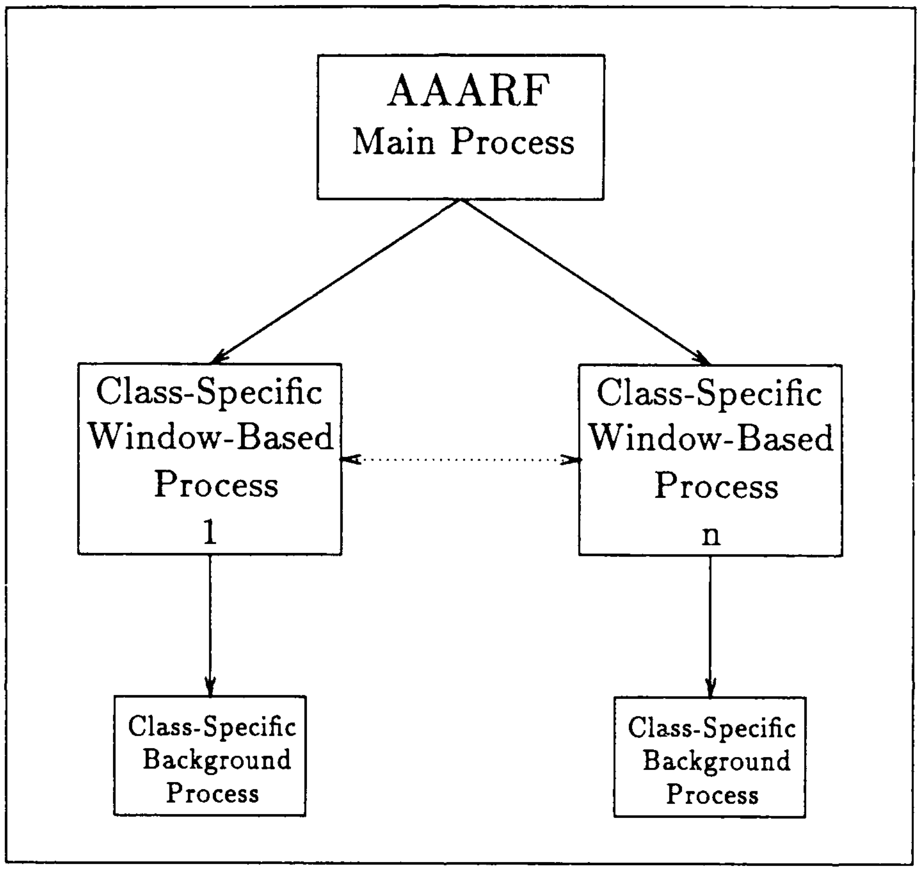


Figure 2: AAARF Levels of Execution

Class-Specific Background Process.

The third level of execution is completely transparent to the user; this level implements the input generator and algorithm. It provides the window-based level with the IEs that drive the animation. The actual algorithms being animated run at this level. The background process waits with the next IE until the window-based level sends an IE request. The background process sends the IE and executes the algorithm until the next IE occurs. It stops again and waits for an IE request from the window-based level.

2.2 Windows

AAARF uses three types of windows to provide the user with multiple simultaneous algorithm animations and multiple views of each animation. Figure 3 shows the types of windows used by AAARF and their relationship to one another.

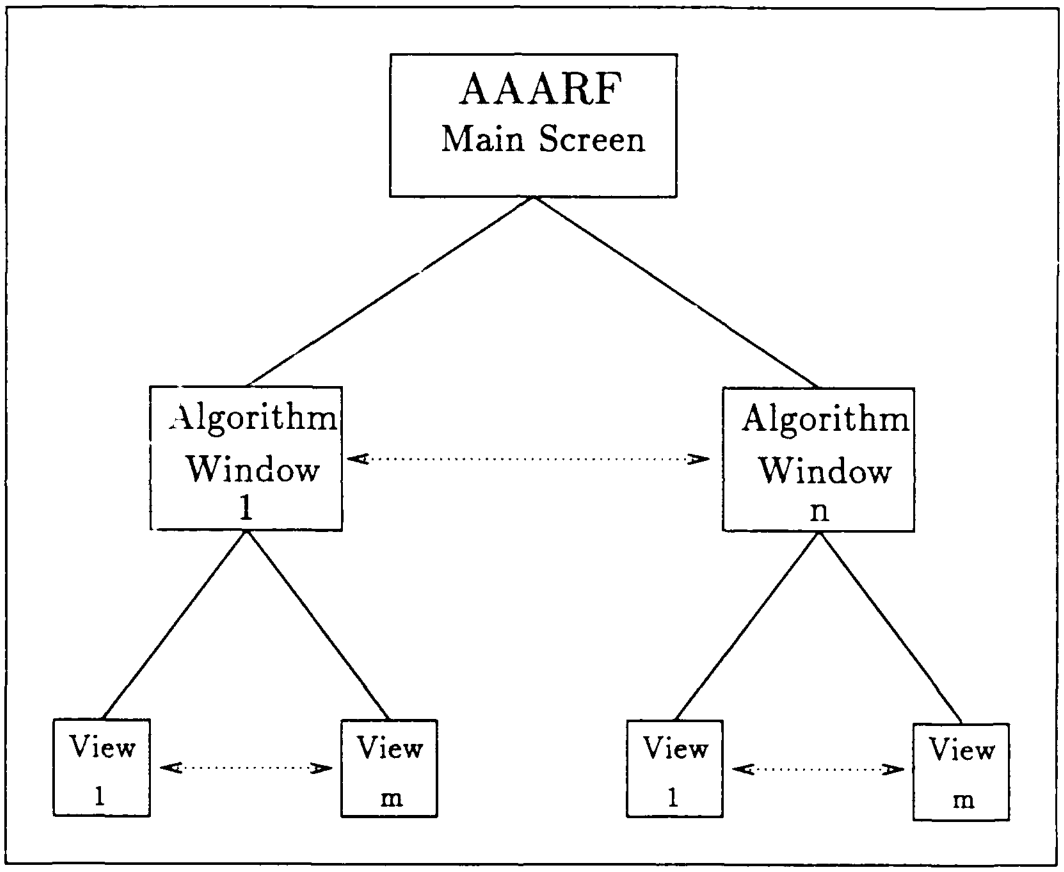


Figure 3: Types of Windows used by AAARF

AAARF Main Screen.

This is a full-screen window within which all interaction with AAARF is contained. The main menu can be *popped up* anywhere on the AAARF main screen. The main screen supports up to four algorithm windows.

- *The actual number of algorithm windows possible is dependent on the system configuration, cpu load, and memory availability.*

Algorithm Window.

All animations take place within algorithm windows. Every algorithm window is associated with a particular algorithm class. Algorithm windows may be created and destroyed freely, but no more than four can be active at any time. Each algorithm window supports an animation recorder, a master control panel, a status display, and up to four view windows. Algorithm windows can be resized and moved anywhere on the AAARF screen; they may overlap one another. The algorithm menu can be popped up anywhere along the algorithm window title bar.

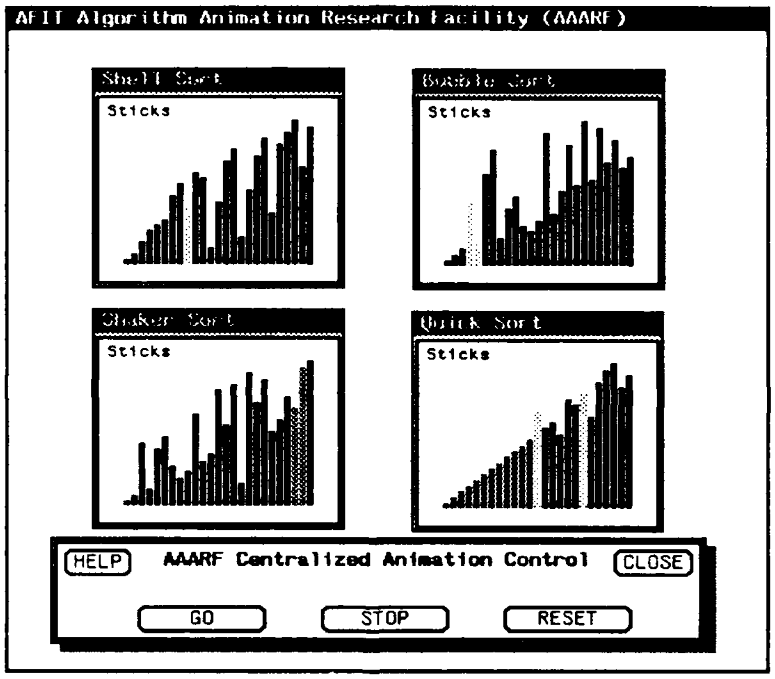


Figure 4: Multiple Algorithm Windows

View Windows.

View windows, or views, are windows associated with a particular view of an algorithm. Every algorithm window has at least one and as many as four active view windows. View windows can be resized and moved, but they cannot extend beyond the algorithm window and they may not overlap.

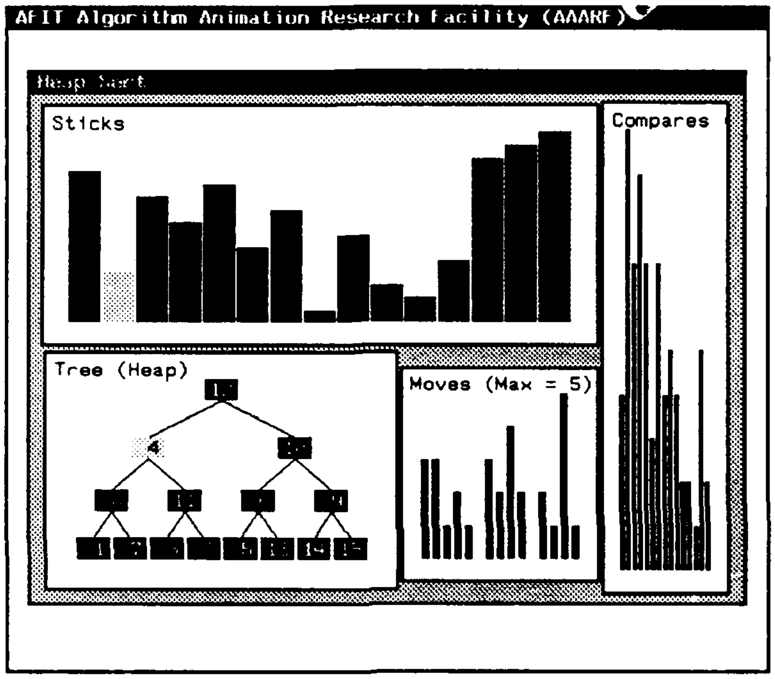


Figure 5: Multiple Views of a Single Algorithm Animation

- Right Button. The right button is used almost exclusively to pop up menus (see Section 2.4) from which selections can be made.
- Middle Button. The middle button is used primarily to position and re-size windows. Figure 9 describes the procedure for moving and resizing windows. The middle button is also used to select an element of interest within a view window by clicking on the desired element.
- Left Button. The left mouse button is used to activate panel items (see Section 2.5). It is also used to control algorithm animations; clicking in a view window starts and stops the animation.

Figure 6: Mouse Button Usage

2.3 The Mouse

Almost all interaction with AAARF is through the mouse. The mouse is used to move the pointer, or cursor, around the screen. The mouse buttons may be either *clicked* (pushed and immediately released) or *depressed* (pushed and held until some action is complete). The function of the mouse buttons depends on the application; Figure 6 describes the mouse button functions. Figure 9 explains how to use the mouse to manipulate windows and panels.

- *Just as you can type ahead of the display with the keyboard, you can "mouse ahead" of the display with the mouse. Depending on the CPU load and the number of active processes, display update can be slow. Rest assured the mouse input will, eventually, be acknowledged. Be careful when mousing ahead – you may be clicking on something that won't be there when the click is serviced.*

- Main AAARF Menu. This menu is accessed by pressing the right mouse button anywhere on the AAARF background screen.
- Algorithm Window Menu. This menu is accessed by pressing the right mouse button anywhere along the title bar of an algorithm window.
- Panel Selectors. These menus are accessed by depressing the right mouse button on panel selectors (Section 2.5) or panel choice items (Section 2.5).

Figure 7: AAARF Menus

2.4 Menus

AAARF uses menus to allow users to make a selection from among several choices. Users *pop up* menus by depressing the right mouse button. Generally, menus remain visible only as long as the right button remains depressed. While a menu is visible and the right mouse button is depressed, moving the cursor over a particular menu entry causes that entry to be highlighted. Releasing the right mouse button with a menu item highlighted *selects* that menu item. Figure 7 describes the three types of menus used in AAARF.

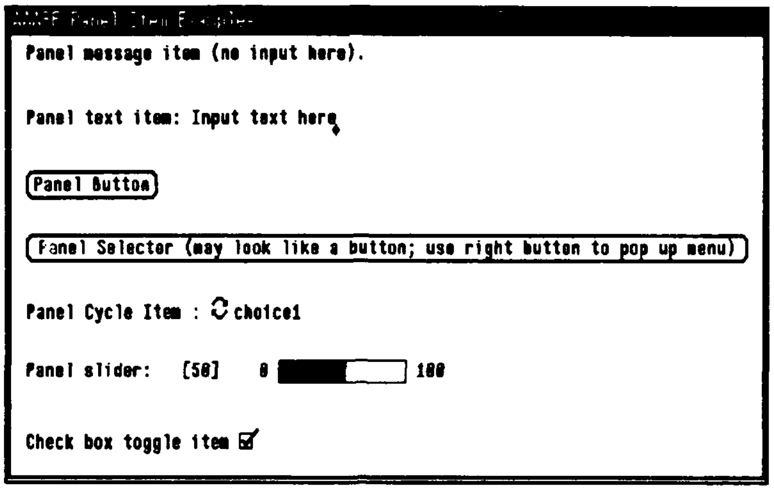
A menu entry with a right-arrow indicates that a *pull-right* menu is associated with that menu item. Moving the cursor over the right-arrow exposes the pull-right menu from which a selection can be made. Usually, the first item in a pull-right menu is the default selection for an item with a pull-right menu.

2.5 Panels

Panels allow users to interact with AAARF through a variety of *panel items* such as, *panel buttons*, *panel selectors*, *choice items*, and *panel text items* (see Figure 8).

Panel Buttons

Panel buttons are used to specify an action. Panel buttons are activated by clicking the left mouse button with the mouse pointer positioned over the panel button.



Panel message item (no input here).

Panel text item: Input text here

Panel Button

Panel Selector (may look like a button; use right button to pop up menu)

Panel Cycle Item : choice1

Panel slider: [50] 0 100

Check box toggle item ☑

Figure 8: Panel Item Examples

Panel Selectors

Panel selectors are used to pop up a menu from which a particular menu item can be selected. Panel selectors are activated by depressing the right mouse button with the mouse pointer positioned over the panel selector. The mouse button is released after the desired menu selection is highlighted.

Choice Items

Choice items are used to set options. Choice items appear as *cycle items*, *choice buttons*, *check boxes*, and *panel sliders*. All choice items except the panel slider can be used as either a panel button or a panel selector. Panel sliders are activated by clicking the left mouse button at the desired position on the slider bar.

Text Items

Text items allow the user to enter data from the keyboard. AAARF requires very little keyboard interaction.

- Moving Windows and Panels. To move a window, position the cursor over the window's border. Press and hold the middle button while repositioning the cursor to the window's new position. Release the middle mouse button and the window moves to the new location. Panels are moved with the same procedure.

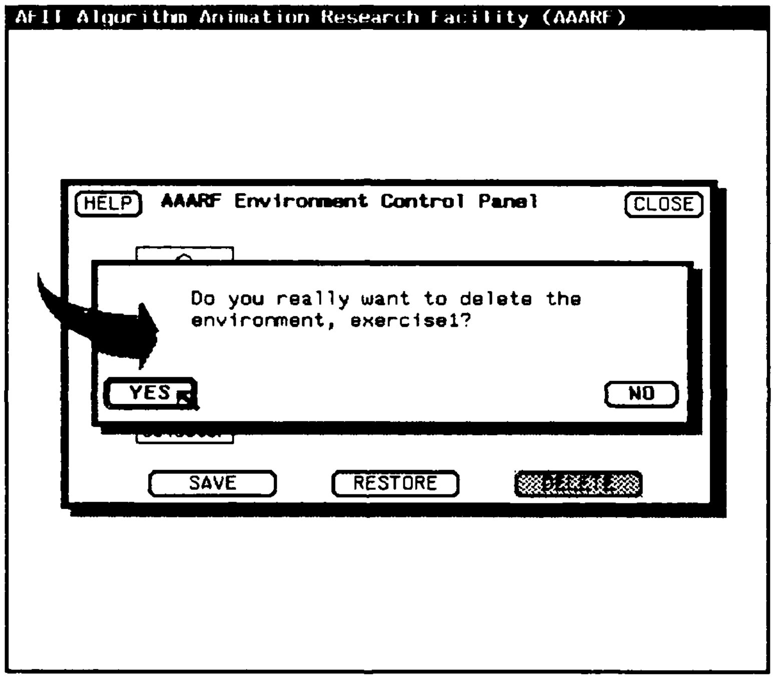
- Resizing Windows. To resize a window, position the cursor over the window's border, press and hold the control key and the middle mouse button while repositioning the cursor to the window's new border position. Release the mouse button and the window resizes. Panels cannot be resized.

- Bringing Windows and Panels to the Front. To completely expose a partially hidden window, click the left mouse button on the hidden window's border. The procedure for panels is identical.

Figure 9: Manipulating Windows and Panels

2.6 Alerts

AAARF uses alerts to display help screens, to warn users of internal errors, and to get confirmation for dangerous commands. The user must click on one of the alert buttons before input is be accepted in any other window or panel. Figure 10 shows an alert asking the user to confirm deletion of an environment file. Note that the **YES** button has a darker outline than the **NO** button. A button with a dark outline indicates that it is the default choice; it may be selected with the mouse or, alternatively, by hitting the return key.


AFIT Algorithm Animation Research Facility (AAARF)

AAARF Environment Control Panel

Do you really want to delete the environment, exercise1?

Figure 10: AAARF Alert Example

3 Getting started

Set your UNIX *path* variable to include the AAARF executable directory. Your instructor or system manager knows the AAARF directory path. See [5] for more information on setting UNIX paths. Run AAARF by entering *aaarf* at the UNIX command line prompt.

3.1 Welcome Screen

The first thing you see is the AAARF welcome screen. Click the left mouse button on the **CONTINUE** button to begin the session.

3.2 The Main Menu

Press and hold the right mouse button with the mouse pointer anywhere on the AAARF screen to pop up the main menu. Move the cursor to select from the following choices:

- Iconify AAARF
- New Algorithm Window
- Central Control
- Environment Control
- Help
- Kill AAARF

Iconify AAARF

This selection causes AAARF and all its animations to become an icon. The icon may be moved anywhere on the screen. While in the icon state, all animations are stopped and most menu items are deactivated. AAARF is deiconified by either clicking the right mouse button on the icon or selecting **Deiconify AAARF** from the main menu.

New Algorithm Window

To open a new algorithm window, highlight this selection. Move the cursor over the right arrow to reveal the algorithm class menu (see Section 3.3). Highlight one of the available classes and release the mouse button. Figure 11 shows the AAARF main menu with the algorithm menu exposed. An algorithm window with a default data set and algorithm appears. Up to four windows may be active simultaneously. See Section 4 for controlling the algorithm window.

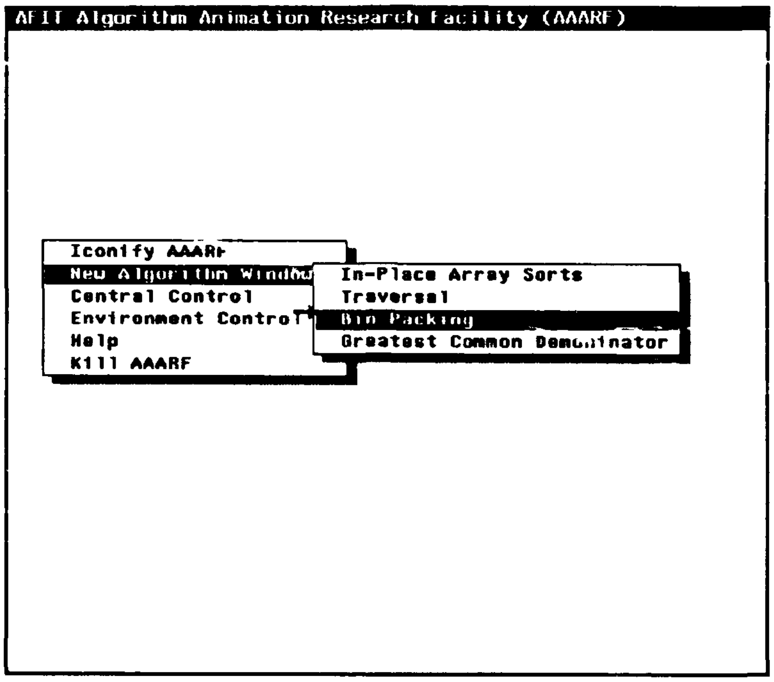


Figure 11: AAARF Main Menu

Central Control Panel

This selection causes the Central Control Panel to appear. The Central Control Panel provides a means to simultaneously control the animations in all open algorithm windows. See Section 3.4 for more information about the Central Control Panel.

Environment Control Panel

This selection causes the Environment Control Panel to appear. The Environment Control Panel provides a means to save and restore AAARF environments. See Section 3.5 for more information about the Environment Control Panel.

Help

This selection displays a help screen which briefly explains the function of each main menu item, how to use the mouse, and how to open a new algorithm window. This help screen is the same as the welcome screen that appears immediately after the program is evoked.

Kill AAARF

This selection ends the AAARF program. The user is first asked to confirm the kill command.

3.3 Algorithm Class File

AAARF has a default set of algorithm classes from which the user can select when opening new algorithm windows. The default classes are named in the *.aaarfClasses* file located with the AAARF executable image. The default algorithm class file can be overridden by setting the **AAARFCLASSES** environment variable to the filename of some other algorithm class file. See [4] for more information regarding UNIX environment variables.

- *The default algorithm class file suffices for most end-users.*

3.4 Central Control Panel

The AAARF Central Control Panel provides a means for simultaneously controlling multiple animations. The usual animation controls provided by the algorithm window are still enabled when using the Central Control Panel. Figure 4 shows the Central Control Panel. Use the right mouse button to initiate the following actions:

- **Help** Displays a help screen explaining the Central Control Pane. unction and controls.

- **Close** Closes the Central Control Panel. The panel can be reopened by selecting it again from the main AAARF menu.

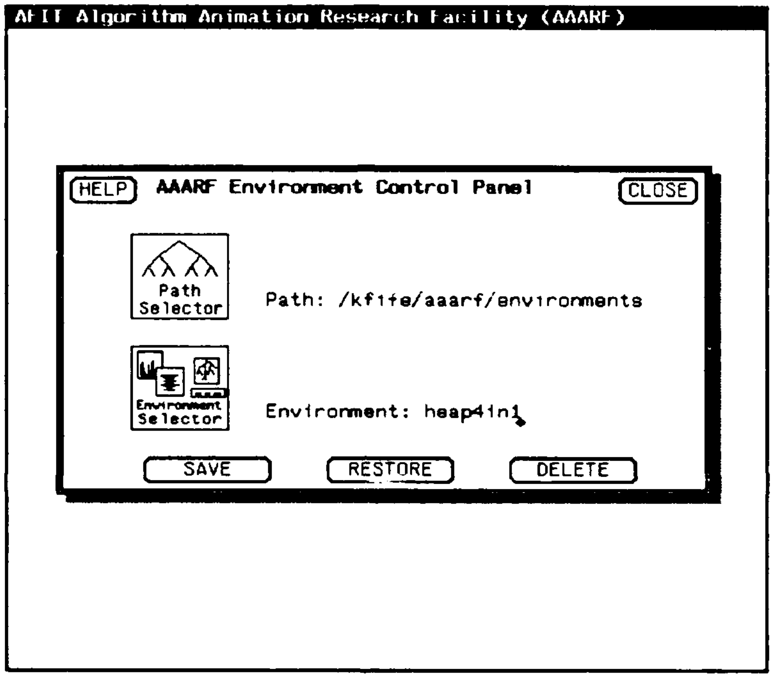
- **Go** Simultaneously starts animations in all open algorithm windows. Animations which have already run to completion are not affected. Use **Reset** followed by **Go** to restart a completed animation

- **Stop** Simultaneously stops animations in all open algorithm windows. Has no affect on animations that have already run to completion

- **Reset** Simultaneously stops and resets all animations to their respective initial conditions. Animations may be running when the **Reset** button is pushed. There is no UNDO for a **Reset**.

3.5 Environment Control Panel

The AAARF Environment Control Panel provides a means for saving the current animation environment or restoring a saved environment. The environment includes *all* user options currently selected: window size, window placement, input parameters, algorithm parameters, and view parameters. The Environment Control Panel is shown in Figure 12.


AFIT Algorithm Animation Research Facility (AAARF)
HELP
AAARF Environment Control Panel
CLOSE
Path Selector
Environment Selector
SAVE
RESTORE
DELETE

Path: /kfife/aaarf/environments

Environment: heap4ini

Figure 12: AAARF Environment Control Panel

Use the **Path Selector** to select a directory in which environments can be saved and restored. The path selector menu is activated by depressing the right mouse button on the Path Selector icon. The path can be changed only by the Path Selector; there is no option to enter the path from the keyboard. By default, the initial path is set to the user's current working directory. The path can be initialized to any valid directory by setting the the **AAARFENV** environment variable to the desired path. Typically users keep all their AAARF environments in a particular directory and set **AAARFENV** to that directory path. See [4] for more information on setting environment variables.

- *To save an AAARF animation environment, the user must have write privileges for the selected directory.*

Use the **Environment Selector** to select an existing environment name, or enter an environment name from the keyboard. Environment names consist of 1 to 15 alphanumeric characters. If a selected directory has no environment files, the **Environment Selector** menu does not pop up.

When the environment name is selected, click on **Save**, **Restore**, or **Delete** to perform the desired operation. Only 100 saved environments are permitted per directory, so it may become necessary to delete unused environments.

- *Save and delete operations are relatively fast, but restore operations can take several seconds depending on the current CPU and memory load.*

The environment control panel is not considered part of the environment with respect to save and restore operations. The panel can be closed only by clicking on the **Close** button.

4 Algorithm Windows

Animations take place in algorithm windows. When an algorithm window is first opened, a default data set, algorithm, and view are provided. These may be changed by opening the master control panel as described later in this section.

Up to four algorithm windows can be active simultaneously. Algorithm windows can be moved, sized, iconified, or destroyed at any time. Algorithm windows contain from one to four non-overlapping view windows for viewing the algorithm in execution.

- *Depending on the CPU load and the number of active processes, it may take several seconds for a new algorithm window to open. Be patient – give the workstation a few seconds to respond before trying again.*

4.1 Algorithm Window Menu

Press and hold the right mouse button with the mouse pointer anywhere on the algorithm window title bar to pop up the algorithm window menu. Figure 13 shows the Algorithm Window Menu. Move the cursor to select from the following choices:

- Iconify
- Master Control
- Animation Recorder
- Status Display
- Help
- Kill

Iconify

This selection causes the algorithm window to become an icon. While in the icon state, the animation is stopped. The algorithm window can be deiconified by clicking on the icon with the left mouse button or selecting **Deiconify** from the algorithm window menu.

Master Control

This selection causes the Master Control Panel to appear. The Master Control Panel provides the user a means for controlling the input, view, algorithm, and control parameters for the animation. Section 4.2 discusses the Master Control Panel in detail.

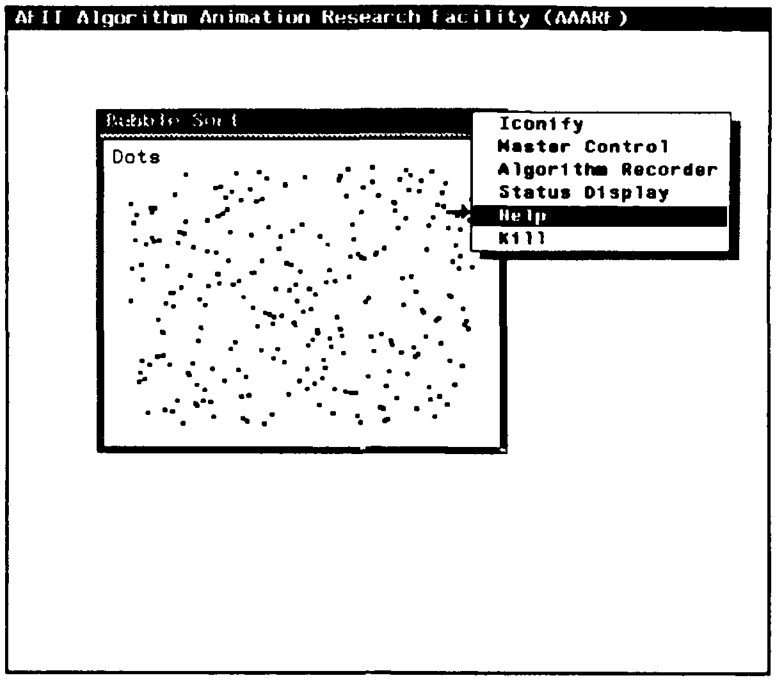


Figure 13: Algorithm Window Menu

Animation Recorder

This selection causes the Animation Recorder to appear. The animation recorder provides a means for creating, saving, and playing animation recordings. Section 4.3 discusses the Animation Recorder.

Status Display

This selection activates the algorithm status display which presents information regarding the algorithm's current state. The middle mouse button can be used to extract more detailed information regarding a particular element or area of interest within the algorithm. The Status Display is discussed in Section 4.4.

Help

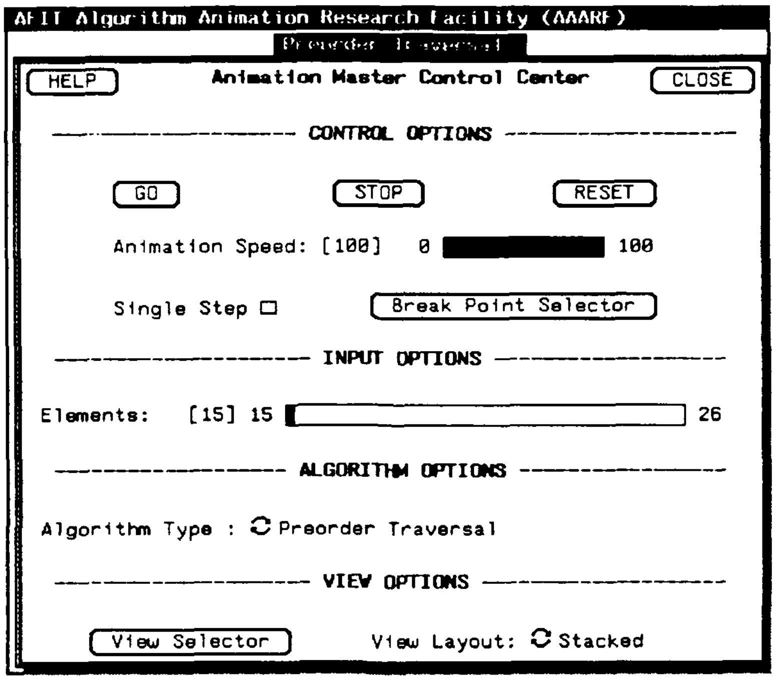
This selection displays a help message which describes each of the options available from the algorithm window menu.

Kill

This selection permanently closes an algorithm window.

4.2 Master Control Panel

Figure 14 shows a typical Master Control Panel. The panel is divided into four separate but interdependent sections: the control section, the input section, the algorithm section, and the view section.



Animation Speed: [100] 0 100

Single Step ☐

Elements: [15] 15 26

Algorithm Type : Preorder Traversal

View Layout: Stacked

Figure 14: Master Control Panel for Traversal Algorithm Class

Control Section

The control section provides panel items for controlling the execution of the algorithm animation. The **Go**, **Stop**, and **Reset** buttons have obvious effects on the animation. To restart an animation which has run to completion, the **Reset** button must be hit. An alternative to these buttons is clicking the left mouse button in any of the open view windows. An algorithm which has run to completion can be reset by clicking the left mouse button in a view window.

The **Single Step** check box, the **Speed** panel slider, and the **Break Point Selector** are three more control mechanisms provided by the control section. Activating single-step mode causes the animation to stop after *every* IE. Selecting one or more break points causes the animation to stop after every IE which matches a selected break point. The speed control affects the execution speed of the animation; 100 is full speed, and 0 is very slow speed.

Input Section

The input section provides parameter controls to produce a variety of input data sets for a particular algorithm class. Changes to input parameters do not take effect until the animation is **Reset**.

Algorithm Section

This section provides, as a minimum, a panel cycle item for selecting the algorithm to be animated. There may be optional parameters for particular algorithms. For instance, quick sort options might be minimum partition size and the method for selecting a pivot value. Changes to algorithm parameters do not take effect until the animation is **Reset**.

View Section

This section provides, as a minimum, a panel selector for selecting from one to four views of the animation, and a panel cycle item for selecting the layout of the views within the algorithm window. There may be optional parameters for controlling the appearance of the view, such as the representation or size of nodes in a binary tree. Changes to view parameters take effect immediately.

4.3 Animation Recorder

Each algorithm window supports an animation recorder for recording and playing back computationally intensive animations. Figure 15 shows the AAARF animation recorder. The recorder also provides a convenient means for saving a particular set of algorithm parameters. Currently the recorder does not support playback in reverse.

The recorder is always in one of three states: OFF, RECORD, or PLAY. The algorithm window title bar reflects the current state of the recorder. Depending on the current recorder state, some operations may not be possible or allowed. Warnings are issued before any recording is erased.

The **Help** button activates a brief help screen explaining how to use the animation recorder. The recorder can be closed only by clicking on the **Close** button.

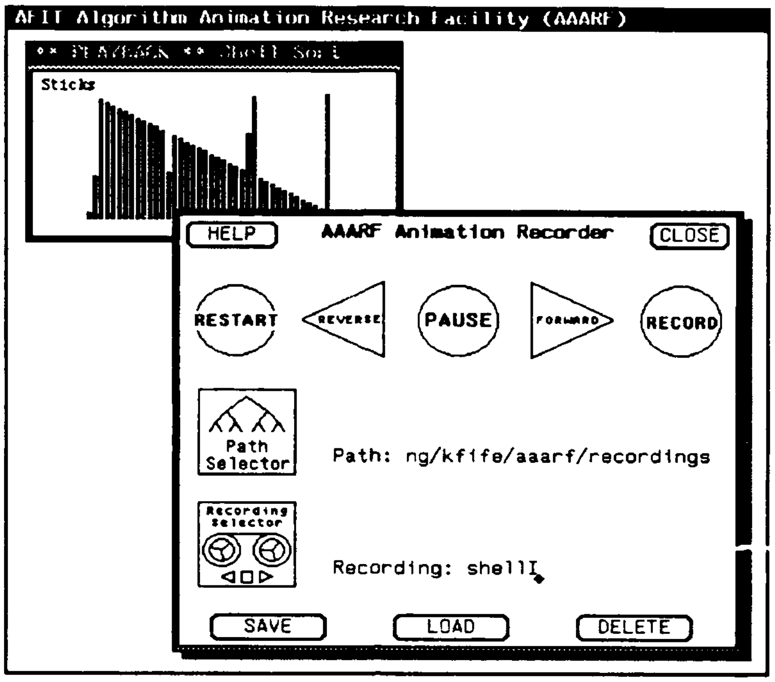


Figure 15: AAARF Animation Recorder

Recording

To record an animation, click on the **RECORD** button. The animation resets using the currently selected parameters, and the algorithm window title bar displays "**** RECORDING ****". Control the animation as usual. When the animation is finished, the recorder switches to PLAY mode and the algorithm window title bar displays "**** PLAYBACK ****". The recording can now be saved or played back.

- *Control commands are not saved; only the interesting events are recorded.*

Playback

Before a recording can be played, it must be recorded. Recordings can be saved, reloaded, and played; or just recorded and played without saving. Once a recording is loaded into the recorder, the algorithm window title bar displays "**** PLAYBACK ****". Playback can be started in several ways: the **FORWARD** button on the recorder, the **GO** button on the Master Control Panel, the **GO** button on the Central Control Panel, or by clicking the left mouse button in a view window. Since only the interesting events are recorded, the view parameters can be changed while the recording is playing back; likewise the recording can be controlled just like a "normal" animation.

File Functions

Use the **Path Selector** to select a directory in which recordings can be saved and restored. The path selector menu is activated by depressing the right mouse button on the Path Selector icon. The path can be changed only by the Path Selector; there is no option to enter the path from the keyboard. By default, the initial path is set to the user's current working directory. The path can be initialized to any valid directory by setting the the **AAARFREC** environment variable to the desired recording directory. Typically, users keep all their AAARF recordings in a particular directory, possibly partitioning it into subdirectories corresponding to each algorithm class. The **AAARFREC** environment variable is set to the parent directory name. See [4] for more information on setting environment variables.

- *To save an AAARF animation recording, the user must have write privileges for the selected directory.*

Use the **Recording Selector** to select an existing recording name, or enter an recording name from the keyboard. Recording names consist of 1 to 15 alphanumeric characters. If a selected directory has no recording files, the **Recording Selector** menu does not pop up.

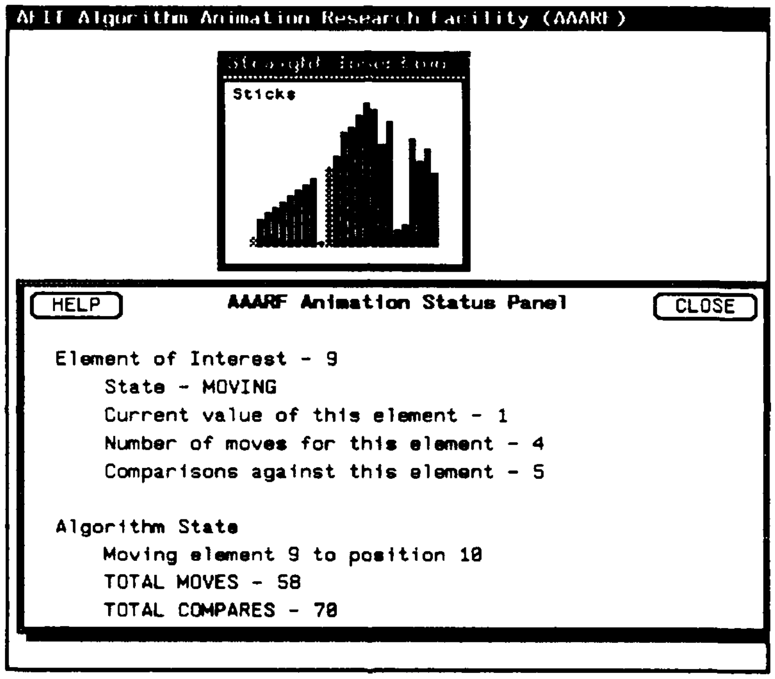
When the recording name is selected, click on **Save**, **Restore**, or **Delete** to perform the desired operation. Only 100 saved recordings are permitted per directory, so it may become necessary to delete unused recordings.

Loading a recording sets the input, algorithm, and control parameters to those associated with the recording; the view parameters are not affected. Saving a recording resets all parameters to their state at the beginning of the recording.

4.4 Status Display

Each algorithm window supports a status display which provides information relating to the current state of the algorithm. Figure 16 shows a typical Status Display. The information and format may be different for each algorithm class. Users can interrogate the animation regarding specific aspects of the algorithm, such as a tree node or array element state by clicking the middle mouse button in a view window on a particular element of interest.

The **Help** and **Close** buttons perform their usual functions.


AFIT Algorithm Animation Research Facility (AAARF)
Sticks
HELP
AAARF Animation Status Panel
CLOSE

```
Element of Interest - 9
     State - MOVING
     Current value of this element - 1
     Number of moves for this element - 4
     Comparisons against this element - 5

Algorithm State
     Moving element 9 to position 10
     TOTAL MOVES - 58
     TOTAL COMPARES - 70
```

Figure 16: Status Display for ArraySort Algorithm Class

5 Exercises

This section presents exercises to test and develop your skill using AAARF. The first exercise directs you through an entire AAARF session. The remaining exercises require a little more thought.

Problem 1. Compare the performance of *Straight Selection Sort* to *Straight Insertion Sort* for sorting an array of 25 numbers. The numbers are already sorted, but in reverse order. Record the animation of the fastest algorithm. How many total moves were required? How many comparisons?

1. Startup AAARF by typing *aaarf* at the UNIX command line prompt.
2. Click on **CONTINUE** to close the welcome screen.
3. Open two ArraySort algorithm windows, by popping up the main menu, selecting the **New Algorithm Window** item, and then selecting **Array Sorts** from the pull-right algorithm menu.
4. Open the Master Control Panels for both windows.
5. Select *Straight Insertion Sort* for one window and *Straight Selection Sort* for the other.
6. Select 25 elements, 100% sorted, and Inverted order for both windows.
7. Select stacked Sticks, Moves, and Compares views for both windows.
8. Set speed to 100% in both windows.
9. Close both Master Control Panels.
10. Open the Central Control Panel.
11. Arrange the algorithm windows and Central Control Panel such that the algorithm windows are the same size, no windows are hidden, and both animations are easy to see and compare. Figure 17 shows a possible arrangement.
12. Open the Environment Control Panel. Select a directory and an environment name, and save this environment.
13. Close the Environment Control Panel.

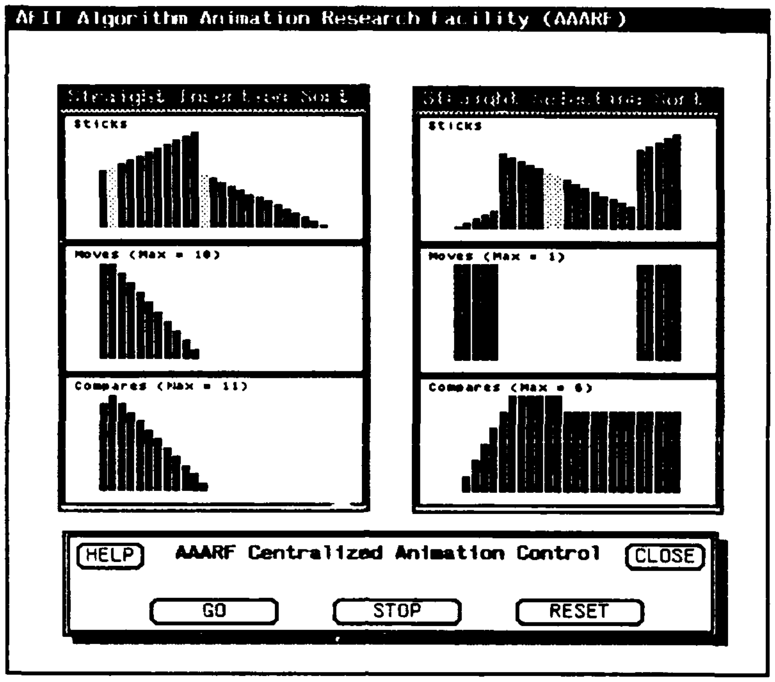


Figure 17: Possible Window Arrangement for Exercise 1.

14. Using the Central Control Panel, **Reset** both algorithm windows.
15. Using the Central Control Panel, start both algorithms by clicking on Go.
16. Determine which algorithm is fastest and close (**Kill**) the other window.
17. Open the Status Display for the fastest algorithm window and note the total comparisons and moves.
18. Close the status display.
19. Open the Animation Recorder.
20. Hit the **Record** button, then hit **Forward**.
21. When the animation is finished, save the recording.
22. Open the Master Control Panel and change the parameter settings. Hit the **Reset** button.
23. Load the recording that was just saved. Notice that the parameter settings reset to those associated with the recording.

24. Playback the recording.

25. **Close** the Recorder and **Kill** the algorithm window.

26. Pop up the AAARF main menu and **Kill AAARF**.

Problem 2. Open three *Traversal* algorithm windows. For each algorithm window, select 26 as the tree size, set the control to single step, and select both the tree and list views, but let the tree view dominate the algorithm window. Select a different algorithm for each window. Arrange and size the algorithm windows so that all three can easily be seen. Open the Central Control Panel and run the animations simultaneously. Save the environment. Which algorithm traverses the entire tree first? Which is last?

Problem 3. A program module to maintain a sorted array of numbers must be developed for the B-10 bomber automatic pilot software. The array of numbers is built by another module from five other sorted lists of numbers. Sometimes the five sub-lists are sorted in ascending order, other times in descending order. The sub-lists are always composed of at least 10 elements. Use AAARF to determine the best sort algorithm for this program module. Create an AAARF environment to compare the top two sort algorithms. Save the animation environment. Which algorithm is probably best?

References

[1] Brown, Marc H. *Algorithm Animation.* Cambridge, Massachusetts: The MIT Press, 1987.

[2] Fife, Keith C. "The AAARF Programmer's Guide." Air Force Institute of Technology, November 1989.

[3] Fife, Keith C. *Graphical Representation of Algorithmic Processes.* MS thesis, Air Force Institute of Technology, 1989.

[4] Sun Microsystems. *Getting Started with SunOS: Beginner's Guide*, 1988. SunOS Technical Documentation.

[5] Sun Microsystems. *Setting Up Your SunOS Environment: Beginner's Guide*, 1988. SunOS Technical Documentation.

[6] Sun Microsystems. *SunView1 Beginner's Guide*, 1988. SunOS Technical Documentation.

Appendix D. *The AAARF Programmer's Guide*

The *AAARF Programmer's Guide* details the implementation of the AAARF program. It provides a guide for developing new animations and maintaining the AAARF program. It is intended to be a stand-alone document.

AAARF Programmer's Guide

Keith Carson Fife
Captain, USAF

Department of Electrical and Computer Engineering
School of Engineering
Air Force Institute of Technology
Wright-Patterson Air Force Base, Ohio 45432

December, 1989

Contents

List of Figures

List of Tables

AAARF Programmer's Guide

Captain Keith C. Fife

1 Introduction

The AFIT Algorithm Animation Research Facility (AAARF) is an interactive algorithm animation system. It provides a means for visualizing the execution of algorithms and their associated data structures. AAARF allows the user to select the type of algorithm, the input to the algorithm, and the views of the algorithm. Several control mechanisms are provided, including stop, go, reset, variable speed, single-step, and break-points. Other features of AAARF include:

- Multiple Algorithm Windows
- Simultaneous Control of Multiple Animations
- Animation Environment Save and Restore Capability
- Multiple View Windows within each Algorithm Window
- Animation Record and Playback
- Algorithm State Display and Interrogation Capability
- Master Control Panel for monitoring and modifying the input, algorithm, view, and control parameters to an algorithm animation.

AAARF runs on Sun3^{TM} and Sun4^{TM} workstations using SunOS^{TM} (Sun Microsystem's version of the AT&T UNIX^{TM} operating system) and the SunView^{TM} window-based environment. AAARF is designed for use with color monitors. It can be used with monochrome monitors, but the displays are not as informative. AAARF is written in the C programming language.

AAARF has two types of users: *end-users* who view and interact with the algorithm animations, and *client-programmers* who develop and maintain the AAARF program and its algorithm animations. **This manual is intended for client-programmers.** It describes the overall AAARF implementation and provides a guide for creating new algorithm animations for AAARF. End-users should refer to the *AAARF User's Manual* [2].

Client-programmers should have experience with AAARF as end-users and be familiar with the terms and concepts associated with algorithm animation. Experience with C, UNIX, and SunView is assumed. The following references are recommended:

- *Getting Started with SunOS: Beginner's Guide* [8]
- *Setting Up Your SunOS Environment: Beginner's Guide* [11]
- *The SunView1 Beginner's Guide* [12]
- *The C Programming Language* [5]
- *C Programmer's Guide* [7]
- *Network Programming* [9]
- *SunView1 Programmer's Guide* [13]
- *Pixrect Reference Guide* [10]
- *Algorithm Animation* [1]
- *The Graphical Representation of Algorithmic Processes* [3]

The next section presents an overview of AAARF and an introduction to terms and concepts associated with algorithm animation; this is the starting point for new AAARF programmers. After gaining some familiarity with AAARF, this section may be used only as a reference. Section 3 discusses the AAARF main process. This section is intended primarily as a maintenance guide for the AAARF main process; however, it is also recommended reading for programmers who are creating new algorithm animations. Section 4 introduces the animation level of AAARF. Aspects of AAARF algorithm animations common to all algorithm classes are presented here. Section 5 discusses the class-specific aspects of creating algorithm animations. Sections 4 and 5 are essential references for the development of new algorithm animations. Section 6 presents some ideas and suggestions for AAARF extensions and programming projects.

2 Overview

This section presents an introduction to several topics of interest to AAARF programmers and animation developers. It outlines some basic concepts of algorithm animation and presents an overview of the AAARF system architecture. The SunView Notifier and program documentation are also discussed. It is important to understand the concepts presented in this section before beginning to develop AAARF applications. See the references listed in Section 1 for more detailed presentations.

2.1 Algorithm Animation Concepts and Definitions

These concepts, extracted from [3] and [1], are critical for maintaining AAARF and its algorithm animations.

Animation Components An algorithm animation consists of three components: an input generator, an algorithm, and one or more animation views (see Figure 1).

Input Generator An input generator is a procedure which provides input to an algorithm; the input may be generated randomly, read from a file, or entered by the user.

Animation View Animation views are graphical representations of an algorithm's execution. The view refers to both the graphical representation and the software that generates the view.

Interesting Event (IE) Animation views are driven by interesting events that occur during the execution of an algorithm. Interesting events are input events, output events, and state changes that an algorithm undergoes during its execution. The type, quantity, and sequence of IEs for a particular algorithm distinguish it from other algorithms.

Algorithm Class Algorithms which operate on identical data structures and perform identical functions are from the same algorithm class. The *Array Sort* class includes *quick sort*, *heap sort*, *bubble sort*, and other in-place sort algorithms. Algorithms from the same class generally share input generators and views, although certain views and input generators may be ineffective with particular algorithms. For instance, the *tree* view is very meaningful for heap sort, but nearly useless for any other sort.

Algorithm Animation System Model The elements of an algorithm animation system are depicted in Figure 2 and include the environment manager, component library, and library manager.

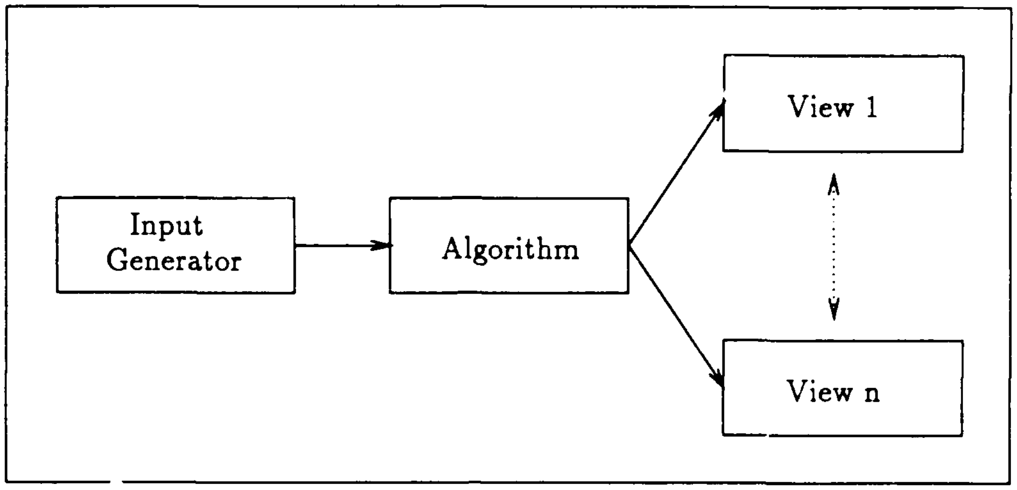


Figure 1: Algorithm Animation Components

Environment Manager The environment manager is the process with which end-users interact to select, control, and view algorithm animations. The environment manager supports multiple algorithm animations and provides a means for controlling their execution.

Component Library The component library is a collection of algorithm components arranged by algorithm class. The environment manager calls routines from the component library.

Library Manager The library manager is the process with which client-programmers interact to maintain the component library. The library manager provides a means to add and delete classes and to create, modify, or delete algorithms, input generators, and views within algorithm classes. Although the environment manager uses the component library directly, changes made by the library manager should not affect the environment manager. The modified components should be immediately ready for use by the environment manager.

Parameterized Control User-selectable parameters are associated with each component. *Algorithm parameters* affect some aspect of how the algorithm executes. For example, with a quick sort algorithm, what partitioning strategy should be used; as the partitions get smaller, at what point should another type of sort be used; what other type of sort should be used. *Input parameters* affect the input generator – what seed is used to generate a set of random numbers; how

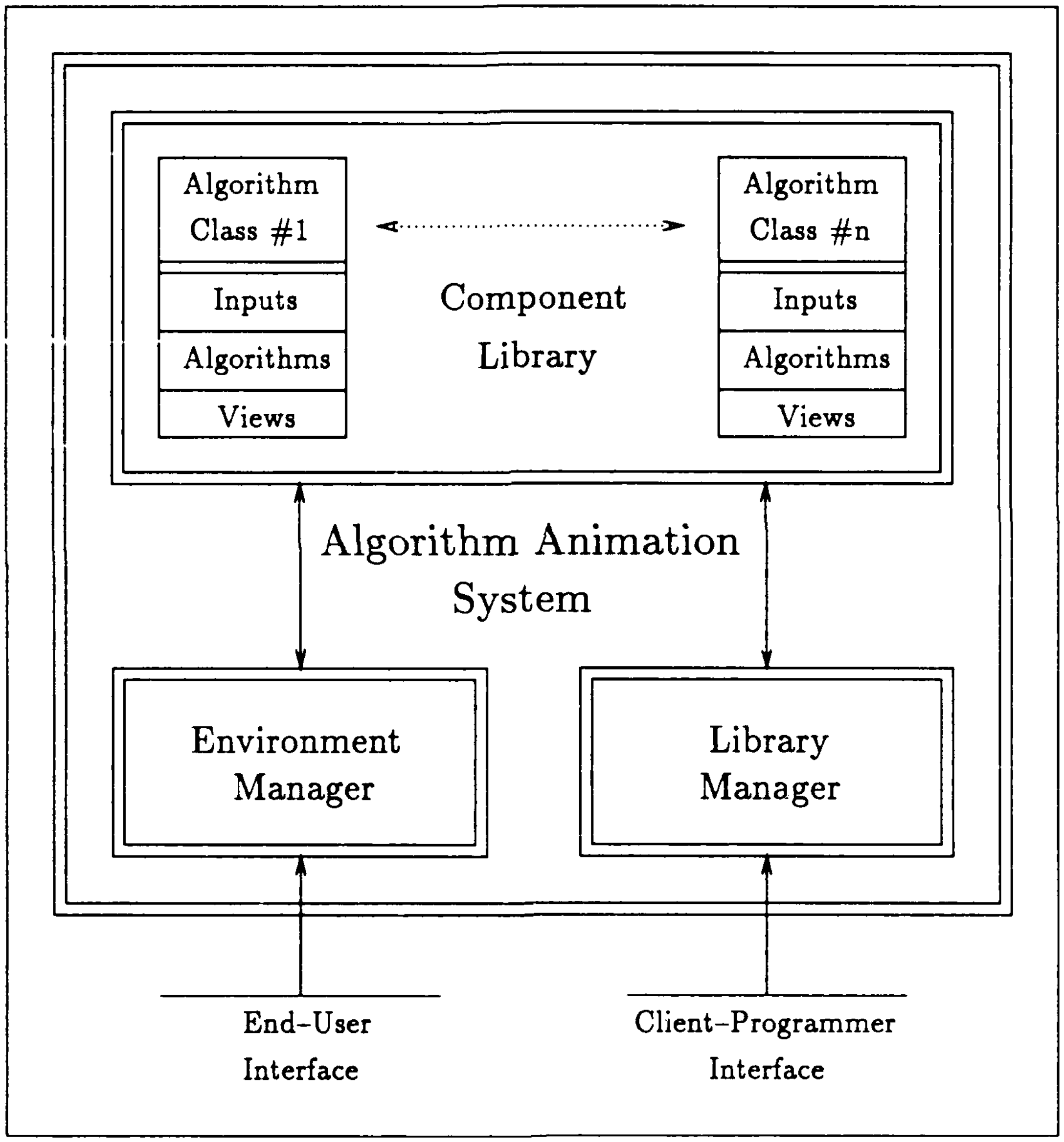


Figure 2: Algorithm Animation System

"sorted" is a set of unsorted numbers; what is the general form of a series of numbers. *View parameters* affect how the animation is displayed in the view window. For example, what shape is associated with the nodes in a graph; how should an arbitrary graph be positioned.

2.2 AAARF Architecture

AAARF uses three levels of execution linked via UNIX sockets [9:191-217] to implement the algorithm animation system model. Sockets are the interprocess communication mechanism provided by the UNIX operating system. Figure 3 shows the levels of execution.

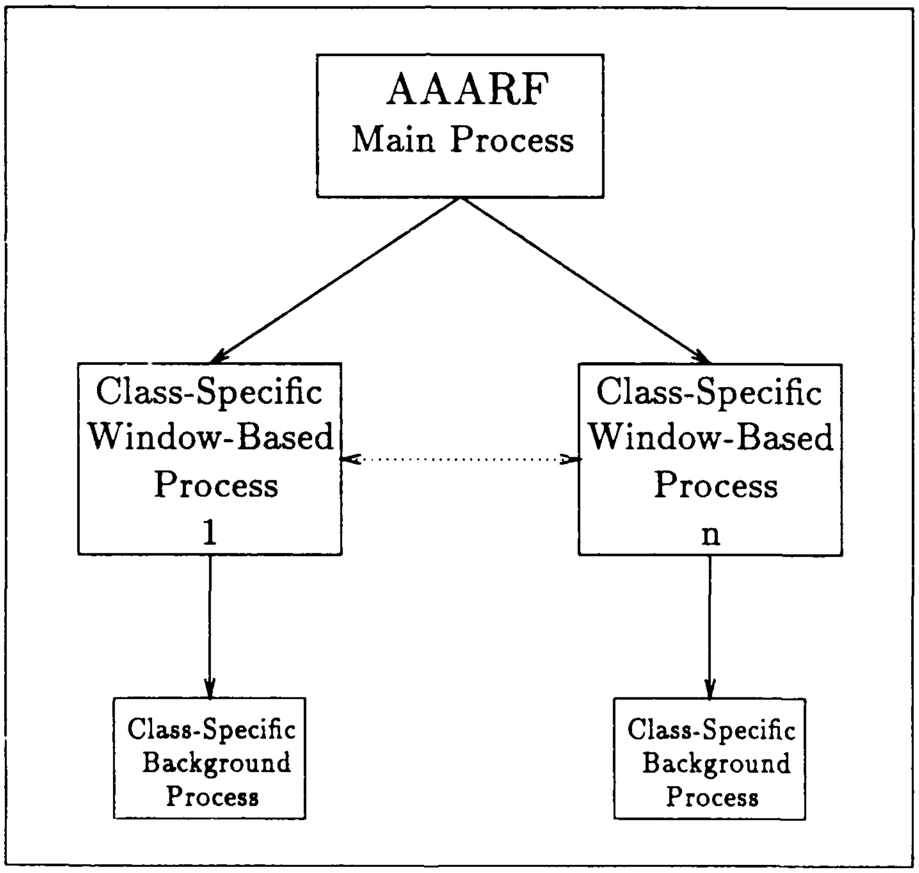

AAARF Main Process
Class-Specific Window-Based Process 1
Class-Specific Window-Based Process n
Class-Specific Background Process
Class-Specific Background Process

Figure 3: AAARF Levels of Execution

The AAARF Main Process

AAARF's top-level process acts as the environment manager. It provides the main screen, a mechanism for saving and restoring animation environments, a mechanism for controlling multiple algorithm animations simultaneously, and a means for starting new algorithm animations.

- *In general, the AAARF main process requires the least attention from client-programmers.*

The Class-Specific Window-Based Process

The second level of execution is the class-specific window-based process. The window-based level of execution provides the animation views, an animation recorder, a status display for interrogating the state of the algorithm, and a master control panel for monitoring and modifying the parameters that affect the animation. The view component of Figure 2 is implemented at this level.

The Class-Specific Background Process

The third level of execution is transparent to the user; this level is the implementation of the input generator and algorithm components of the algorithm animation system model (Figure 2). It provides the window-based level with the IEs that drive the animation. The background process waits with an IE until the window-based level sends an IE request. The background process sends the IE, then resumes execution of the algorithm. At the next IE, the background process again stops and waits for an IE request from the window-based process.

2.3 Windows

AAARF uses three types of windows to provide the user with multiple simultaneous algorithm animations and multiple views of each animation. Figure 4 shows the types of windows used by AAARF and their relationship to one another.

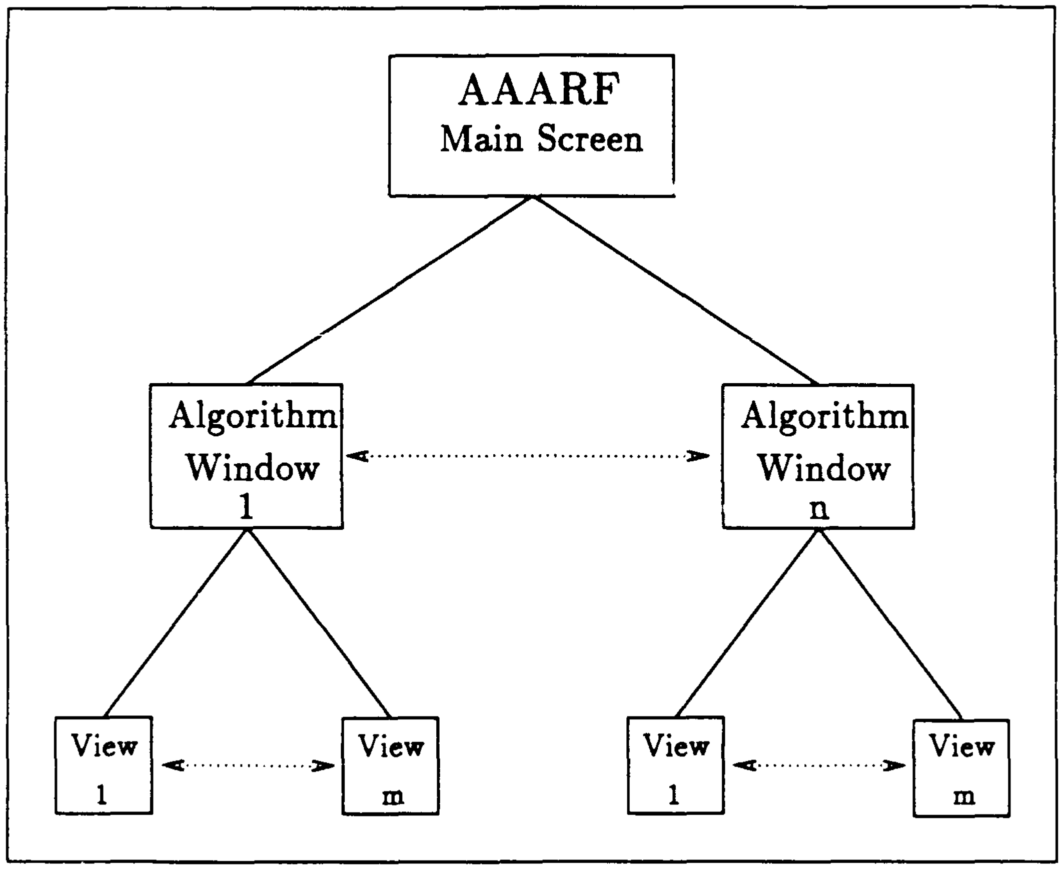


Figure 4: AAARF Windows

AAARF Main Screen

This is a full-screen window within which all interaction with AAARF is contained. The main screen currently supports up to four algorithm windows.

- *The actual number of algorithm windows possible is dependent on the system configuration, cpu load, and memory availability.*

Algorithm Window

All animations take place within algorithm windows. Every algorithm window is associated with a particular algorithm class. Algorithm windows may be created and destroyed at any time, but no more than four can be active at any time. Each algorithm window supports an animation recorder, a master control panel, a status display, and up to four view windows. Algorithm windows can be resized and moved anywhere on the AAARF screen; they may overlap one another.

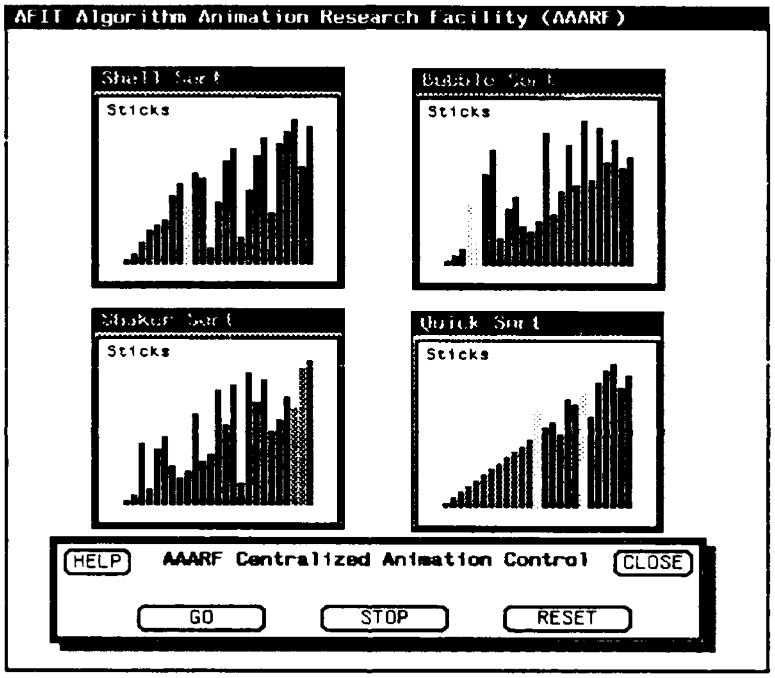


Figure 5: Multiple Algorithm Windows

View Window

View windows, or views, are windows associated with a particular view of an algorithm. Every algorithm window has at least one and as many as four active view windows. View windows can be resized and moved, but they cannot grow outside of the algorithm window and they may not overlap.

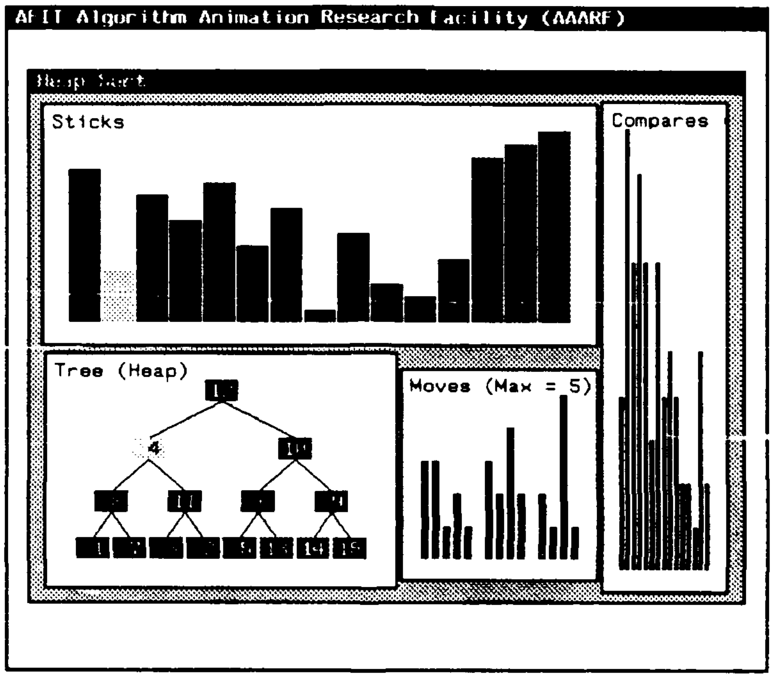


Figure 6: Multiple Views within an Algorithm Window

2.4 AAARF Directory Structure

The AAARF directory structure consists of the *aaarf* parent directory and four subdirectories: *bin*, *main*, *lib*, and *icons*. Any number of class subdirectories may also be included, but there is no requirement for them to exist within the *aaarf* directory.

The *bin* subdirectory contains the aaarf executable image and the default AAARF class file (*.aaarfClasses*). The AAARF classes file lists the available algorithm classes and the name of the executable file for each class. Section 3.2 addresses the AAARF class file in more detail. Typically, algorithm class executables are also stored in *bin*, but it is not a requirement.

The *main* subdirectory contains source code for the main AAARF process. Section 3 discusses the files in this subdirectory.

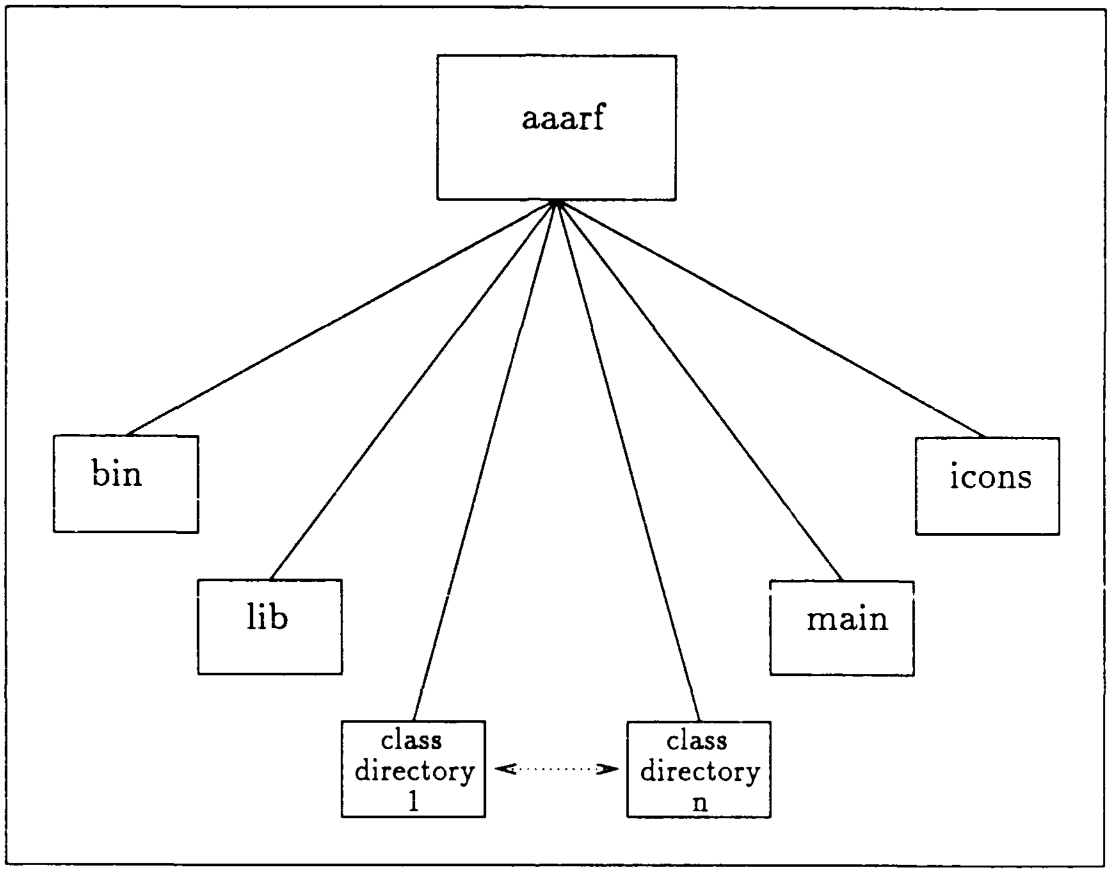


Figure 7: AAARF Directory Structure

The *lib* subdirectory contains source code common to all algorithm classes. Client-programmers use this source code as a starting point for building new algorithm animations. Section 4 discusses the files in this subdirectory.

The *icons* subdirectory contains the icons used by AAARF.

If class subdirectories exist, they contain the source code for their respective algorithm class executables. Section 5 discusses the functions required for creating an algorithm animation.

2.5 SunView

SunView is an object-oriented system that provides a set of visual building blocks for assembling user interfaces. SunView objects include windows, pointers, icons, menus, alerts, panel items, and scrollbars. AAARF makes extensive use of SunView objects;

only the background process does not use SunView.

SunView is a ***notification-based*** system. The Notifier acts as the controlling entity within an application, reading UNIX input from the kernel, and formatting it into higher-level ***events***, which it distributes to the appropriate SunView objects. The Notifier ***notifies***, or calls, various procedures which the application has previously registered with the Notifier. These procedures are called *notify procedures*. The SunView Notifier model is shown in Figure 8.

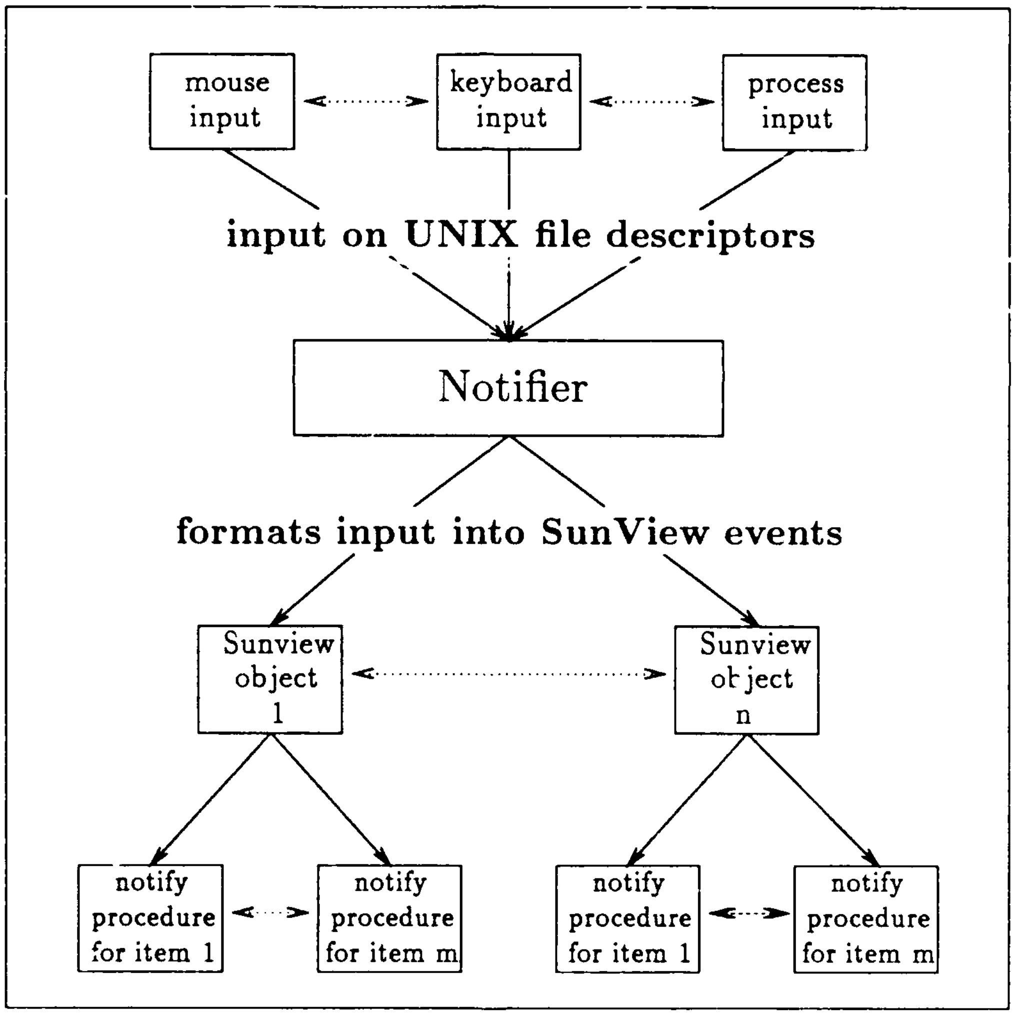

mouse input
keyboard input
process input
input on UNIX file descriptors
Notifier
formats input into SunView events
Sunview object 1
Sunview object n
notify procedure for item 1
notify procedure for item m
notify procedure for item 1
notify procedure for item m

Figure 8: SunView Notifier Model [13:23]

2.6 Program Documentation

The existing source code is fully documented in accordance with AFIT System Development Documentation Guidelines and Standards [4]. Most functions are less than one page long and most modules contain less than ten functions. The existing AAARF source code should be used along with this manual in developing new animations.

- FILE : The module name (UNIX file name)
- PROJECT : AFIT Algorithm Animation Research Facility (AAARF)
- DATE : Date of current version number
- VERSION : Current version number
- AUTHOR : Person or persons who are responsible for the module.
- DESCRIPTION : Description of the module's function.
- FUNCTIONS : List of functions implemented in the module.
- REFERENCES : References which could help explain what's happening in the module.
- HISTORY : List of major changes to the module

Table 1: Module Header Information

- FUNCTION NAME : Name of the function.
- FUNCTION NUMBER : Function's number within this module.
- VERSION NUMBER : Current version number.
- DATE : Date of current version number.
- DESCRIPTION : Detailed description of what the function does.
- INPUT PARAMETERS : List and description of input parameters.
- OUTPUT PARAMETERS : List and description of output parameters.
- GLOBALS USED : List of global variables used.
- GLOBALS AFFECTED : List of global variables changed.
- FUNCTIONS CALLED : List of functions called.
- HISTORY: List of major changes to the function.

Table 2: Function Header Information

2.7 Client-Programmer Tasks

Client-programmers are most frequently concerned with the window-based level and its corresponding background process. Undoubtedly, their most difficult task is identifying IEs and developing meaningful views to represent them. Other typical tasks include:

- Adding new input generators and modifying existing input generators for an existing algorithm class.
- Adding new algorithms and modifying existing algorithms for an existing algorithm class.
- Adding new views and modifying existing views for an existing algorithm class.
- Developing new algorithm classes.
- Improving and extending capabilities of main AA/ RF process.

3 The AAARF Main Process

The AAARF main process is not associated with any particular algorithm animation, but provides an environment in which *any* algorithm animation can be controlled. The AAARF main process is stand-alone, but without animation processes to run, it is not very interesting. The source code for the AAARF main process is contained in the *main* subdirectory and consists of the following files:

- *aaarf.c* Creates the main AAARF screen and displays the welcome screen.
- *aaarfControl.c* Creates and maintains the Central Animation Controller.
- *aaarfEnvironment.c* Creates and maintains the AAARF Environment Control Panel.
- *aaarfMenu.c* Creates and maintains the AAARF main menu.
- *aaarfWindows.c* Opens and closes algorithm windows. Maintains accounting information regarding open algorithm windows.

Each module has a corresponding header file in which the module's functions, globals, and data structures are declared. The following sections discuss each of the five modules.

3.1 aaarf.c

This is the top level module for the AAARF main process. It creates the AAARF main screen and calls functions from the other modules to create their respective contributions to AAARF. Table 3 lists the functions included in *aaarf.c*.

The `aaarfEventDispatch()` function interposes user input to AAARF such that:

- The cursor's screen position is saved when the right mouse button is depressed. Windows and panels appear at the most recently recorded cursor position.
- If AAARF is iconic, the left mouse button deiconifies AAARF. Otherwise the left mouse button is ignored, thus disabling its usual function of popping and pushing windows.
- If AAARF is iconic, the middle mouse button is used to move the icon. Otherwise, the middle button is ignored, forcing the AAARF main screen to be either full-size or iconic. This eliminates any problems with hidden or lost algorithm windows among other application windows.
- Ignore ACTION keys. Forces use of mouse and eliminates another way to loose windows.

A polling timer is associated with `aaarfShowWelcomeMessage()` in `main()`. When `window_main_loop()` is entered, the Notifier calls `aaarfShowWelcomeMessage()`. It displays the AAARF help screen and disables its polling timer. Occasionally, on a Sun3, the help screen is displayed before the AAARF main screen - the solution has been to set the polling timer for a longer interval.

int	main(int, char**)
Notify_value	aaarfEventDispatch(Frame, Event, Notify_arg, Notify_event_type)
Notify_value	aaarfShowWelcomeMessage(Notify_client, int)
void	aaarfHelp()

Table 3: Function Prototypes for *aaarf.c*

3.2 aaarfMenu.c

This module creates the main menu. In doing so, it registers a notify_procedure for each menu item with the Notifier. When a menu item is selected, the Notifier evokes the appropriate function. Table 4 lists the functions implemented in *aaarfMenu.c*.

At startup, `main()` calls `setupMainMenu()` which looks for the AAARFCLASSES environment variable. If it's set to a valid AAARF class file, that file is used to generate the algorithm class menu. Otherwise, the default AAARF class file, *.aaarfClasses*, is used.

The AAARF class file contains a list of algorithm class names and their corresponding executable image file names. `setupMainMenu()` reads the class names into the `classStrings` array used to define menu items. It reads the executable filenames into the `classPrograms` array used by *aaarfWindows.c* to execute the animations.

void	setupMainMenu()
caddr_t	menuHelp(Menu, Menu_item)

Table 4: Function Prototypes for *aaarfMenu.c*

3.3 aaarfControl.c

This module creates and manages the central control panel. Table 5 lists the functions implemented in *aaarfControl.c*.

`setupCentralControl()` registers `centralControlDispatch()` with the Notifier. `openControlPanel()` is registered by `setupMainMenu()` in *aaarfMenu.c*.

`centralControlDispatch()` handles most panel events internally, but calls `closeControlPanel()` and `controlHelp()` to close the panel and display a help screen. It uses `aaarfAnnounce()` to send control commands to the active algorithm animations.

void	setupCentralControl()
void	centralControlDispatch(Panel_item, struct inputevent*)
caddr_t	openControlPanel(Menu, Menu_item)
void	closeControlPanel()
void	controlHelp()

Table 5: Function Prototypes for *aaarfControl.c*

3.4 aaarfEnvironment.c

This module creates and manages the AAARF environment save and restore feature. The functions implemented in this module are listed in Table 6.

In `setupEnvironmentPanel()`, the initial AAARF environment path is set to the value of the AAARFENV environment variable, if it exists and is a valid directory. Otherwise the initial path is set to the user's current working directory.

`setupEnvironmentPanel()` registers `environmentHelp()`, `buildMenu()`, `environmentDispatch()`, `closeEnvironmentPanel()`, and `getSelection()` with the Notifier. `createMainMenu()` registers `openEnvironmentPanel()`. The file and environment selectors are associated with a menu. `buildMenu()` is associated with each of the menus and `getSelection` is associated with each of the selectors.

Since the paths and files are constantly changing, the path and file menus cannot reliably be built until immediately before they are displayed. When the path or menu selector icons are activated, the `buildMenu()` function is called to create the appropriate menu. First, all the old menu items are destroyed. Then the new menu items are created from the current `pathNames` and `fileNames` lists.

`pathNames` and `fileNames` are built by calls to `getSubdirectories()` and `getDirectory()`. The name lists are built when the environment panel is opened and after every file operation.

The `environmentDispatch()` function handles all the file transactions and error checking.

void	setupEnvironmentPanel()
void	environmentDispatch(Panel_item, struct inputevent*)
caddr_t	openEnvironmentPanel(Menu, Menu_item)
void	closeEnvironmentPanel(Panel_item, struct inputevent*)
Menu	buildMenu(Menu, Menu_generate)
int	getSelection(Panel_item, Event*)
void	environmentHelp(Panel_item, struct inputevent*)

Table 6: Function prototypes for *aaarfEnvironment.c*

3.5 aaarfWindows.c

This module opens, closes, and monitors algorithm windows. Communication with the algorithm processes is via UNIX sockets. The ALG_WINDOW and AAARF_STATE data structures are used to manage the algorithm windows. Table 7 lists the functions implemented in this module and Table 8 presents the data structure definitions.

createMainMenu() registers openAlgorithmWindow(), aaarfIconify(), and aaarfKill() with the Notifier. aaarfChildMonitor() is registered as the termination notify_procedure for each algorithm window process.

Both openAlgorithmWindow() and restoreEnvironment() fork algorithm window processes. They get a socketpair, fork a process, send the process some setup data, and update aaarfState.

aaarfChildMonitor() is notified each time a child process terminates. It updates the aaarfState. This function is *not* called when a child process is killed (UNIX kill() function), *provided* the function that kills the child waits (UNIX wait() function) for the child process to terminate.

AAARF, all AAARF panels, all open algorithm windows, and all their open panels are iconified and deiconified by aaarfIconify().

aaarfAnnounce() provides a channel for sending commands to all algorithm processes. *Typical commands are* ICONIFY, GO, STOP, *and* RESET. *It always waits* for an acknowledgment.

saveEnvironment() and restoreEnvironment() are the only other functions that communicate directly with the algorithm window process. They also write and read environment parameters to and from disk files.

killWindow() is used to kill algorithm window processes. It uses the UNIX kill() function. After killing a process it uses the UNIX wait4() function to wait for the process to terminate.

void	setupAlgorithmWindows()
caddr_t	openAlgorithmWindow(Menu, Menu_item)
int	aaarfAnnounce(int)
Notify_value	aaarfChildMonitor(Notify_client, int, union wait*, struct rusage*)
Menu_item	aaarfIconify(Menu_item, Menu_generate)
void	saveEnvironment(int)
void	restoreEnvironment(int)
caddr_t	aaarfKill(Menu, Menu_item)
void	killWindow(int)

Table 7: Function Prototypes for *aaarfWindows.c*

```
typedef struct
    {
        int open;
        int pid;
        int socket;
        int class;
    }
    ALG_WINDOW;

typedef struct
    {
        ALG_WINDOW algWindow[MAX_ALG_WINDOWS];
        int numberOpenWindows;
        int CCopen;
        int CCx;
        int CCy;
    }
    AAARF_STATE;
```

Table 8: *aaarfWindows.c* Data Structures

4 Class-Common Library

Each algorithm class that AAARF calls is actually a stand alone process. Without the communication links to AAARF, the process could be executed without the AAARF main process. Every algorithm class process has the following common features:

- Main function gets setup parameters from AAARF,
- Creates base window for animation,
- Creates the master control panel,
- Creates the algorithm window menu,
- Creates the four animation canvases (views),
- Creates the animation recorder,
- Creates the status display,
- Runs the background process,
- Paints the active canvases,
- Enters the SunView main loop.

The AAARF class-common library presents client-programmers a framework for developing new algorithm animations by providing all the common functions. The client-programmer simply develops the class-specific input, algorithm, view, and control functions. Because the class-common library depends on some class-specific data structures, linkable object code is not provided, only source code. The library must be recompiled for each new algorithm class. This limitation could be eliminated in a future version of AAARF.

Not only does the AAARF class-common library make animation development easier for client-programmers, it also forces a consistent animation interface for the end-user. Every algorithm class supports the same set of tools, and those tools work identically for every algorithm class. After animating one algorithm, end-users can animate any algorithm, regardless of the algorithm class or its client-programmer. The class-common library also ensures a consistent communications interface with the AAARF main process.

The source code for the AAARF class-common library is in the *lib* subdirectory and consists of the following files:

- *aaarfCommon.c* Creates the base window for the algorithm class. Creates the algorithm window menu and the animation canvases. It also handles some other miscellaneous initialization tasks.

- *aaarfMaster.c* Creates the master control panel and sets up the control section of the panel.

- *aaarfRecorder.c* Creates the animation recorder. It also provides the primary interface to AAARF.

- *aaarfViews.c* Sets up the view section of the master control panel and creates the status display. Handles the view resizing and repainting and manages the animation.

- *aaarfUtilities.c* This is a set of functions used by the AAARF main process as well as the animation processes.

Each module has a corresponding header file in which the module's functions, globals, and data structures are declared. Each of the five modules are discussed in the following sections.

In addition to the module header files, the following header files define constants and data structures that are shared between two or more modules:

- *aaarfDefines.h* Defines system-wide constants used by all levels of AAARF. This header file is included in all AAARF source code modules. Table 9 lists the *aaarfDefines.h* header file.

- *aaarfIPC.h* This file defines interprocess communication (IPC) constants and data structures shared by the AAARF main process and the class-common library modules. This header file is listed in Table 10.

- *commonDefines.h* Defines constants and data structures shared by all modules within the class-common library. This header file appears as Table 11.

- *classCommon.h* This file defines constants and data structures defined by the class-common library and shared with the class-specific modules. Table 13 lists the *classCommon.h* header file.

- *classSpecific.h* This header file is defined by the class-specific modules and shared with the class-common modules. It requires a specific set of defines and data structures. It is discussed in Section 5 and listed in Table 20.

```
#define MAX_CLASSES                                  50
#define MAX_ALG_WINDOWS                               4
#define MAX_VIEWS                                     4
#define MAX_AAARF_NAME_SIZE                          15
#define MAX_FILES                                   100
#define MAX_FILENAME_SIZE                           256
#define MAX_PATHS                                   100
#define MAX_PATHNAME_SIZE                          1024
#define DISPLAYED_PATH_LENGTH                        25
#define MAX_COMMAND_LINE_SIZE                      1124
#define AAARF_MINIMUM_WIDTH                         500
#define AAARF_MINIMUM_HEIGHT                        500
#define MAX_STRING_LENGTH                            40
#define ASCII_INT_LENGTH                             20
#define MIN_WINDOW_WIDTH                            150
#define MIN WINDOW_HEIGHT                           150
#define OFF                                           0
#define ON                                            1
#define FALSE                                         0
#define TRUE                                          1
#define STOP                                          0
#define GO                                            1
#define CHILD                                         0
#define PARENT                                        1
#define DOWN                                          0
#define UP                                            1
#define FAILURE                                       0
#define SUCCESS                                       1
#define DEFAULT_SLIDER_WIDTH                        256
#define MAX_RAND                             2147483647

#define MAX(a, b)                       ((a > b)?a : b)
#define MIN(a, b)                       ((a < b)?a : b)
#define ABS(a)                       ((a > 0)?a : a * -1)
```

Table 9: *aaarfDefines.h*

```
#define NEW_WINDOW                         -1
#define ICONIFY                           100
#define DEICONIFY                         101
#define SIMULTANEOUS_GO                   200
#define SIMULTANEOUS_STOP                 201
#define SIMULTANEOUS_RESET                202
#define ENV_SAVE                          300
#define SEND_DATA                         301
#define ACKNOWLEDGE                       400

typedef struct
    {
        int width;
        int height;
        int xPos;
        int yPos;
        int color;
    }
    SETUP_PACKET;
```

Table 10: *aaarfIPC.h*

```
#define P_VIEWS                  0
#define P_LAYOUT                 1

#define P_OPEN                   2
#define P_WIDTH                  0
#define P_HEIGHT                 1
#define P_X_POS                  2
#define P_Y_POS                  3
#define P_COLOR                  4
#define P_M_OPEN                 5
#define P_M_X_POS                6
#define P_M_Y_POS                7
#define P_R_OPEN                 8
#define P_R_X_POS                9
#define P_R_Y_POS               10
#define P_S_OPEN                11
#define P_S_X_POS               12
#define P_S_Y_POS               13

#define P_BREAKPOINTS            0
#define P_SINGLESTEP             1
#define P_SPEED                  2

#define AUTO_RESIZE             10
#define USER_RESIZE             20
#define PROP_RESIZE             30
#define LAYOUT_STACKED           0
#define LAYOUT_SIDE_BS           1
#define LAYOUT_CORNERS           2
#define LAYOUT_FILL              3

#define ALG_STOP                 0
#define ALG_GO                   1
```

Table 11: *commonDefines.h* (1 of 2)

```
#define ALG_RESET                      2
#define ALG_TOGGLE                     3
#define ALG_FINISHED                   4
#define ALG_QUERY                      5
#define ALG_WIN_OPEN                   6
#define ALG_WIN_CLOSE                  7

#define REC_OFF                        0
#define REC_RECORD                     1
#define REC_PLAY                       2
#define REC_RESTART                    3
#define REC_REVERSE                    4
#define REC_FORWARD                    5
#define REC_QUERY                      6
#define REC_AT_HEAD                    7
#define REC_AT_NEXT                    8
#define REC_AT_TAIL                    9
#define REC_FINISHED                  10
#define REC_STOP                      11
#define REC_GO                        12

#define T_OFF                          0
#define T_ON                           1
#define TIMER_NULL        (( struct itimerval *)0)

typedef struct
    {
        PARAMS input;
        PARAMS algorithm:
        PARAMS view;
        PARAMS control;
        PARAMS window;
    }
    RESTORE_PAK;
```

Table 12. *commonDefines.h* (2 of 2)

```
#define ANIM_FINISHED                                    5
#define ANIM_BREAK                                       3
#define ANIM_CONTINUE                                    1

typedef int PARAMS[PARAM_SIZE];

typedef struct
    {
        PARAMS input;
        PARAMS algorithm;
    }
    INIT_PAK;

typedef struct
    {
        Canvas canvas;
        Pixwin *pixwin;
        int open;
        int view;
        int width;
        int height;
        int xPos;
        int yPos;
    }
    VIEW_WINDOW;

typedef struct
    {
        VIEW_WINDOW window[MAX_VIEWS];
        int numberOpenViews;
    }
    VIEW_STATE;
```

Table 13: *classCommon.h*

4.1 aaarfCommon.c

This is the top level module for the class-specific window-based process. It creates the base algorithm window, the algorithm window menu, and the animation canvases on which the views are displayed. Table 14 shows the functions included in this module.

`main()` creates the base algorithm window and registers `frameInterposeDispatch()` to monitor window events. It calls several functions to create the standard set of algorithm window tools. `aaarfCommandDispatch()` is registered to monitor the UNIX socket to the AAARF main process.

`frameInterposeDispatch()` catches events in the base algorithm frame before they are dispatched to the appropriate function. Most events are accepted. If a resize event is detected, the old and new sizes are recorded and the window's current views are repainted with calls to `resizeAllViews()` and `repaintAllViews()`.

`aaarfCommandDispatch()` monitors a socket to AAARF. When commands are received, the appropriate functions are called.

The algorithm window is iconified and deiconified in `setAlgWindowState()`.The state of the animation and the state and position of the algorithm panels are preserved.

`createAnimationCanvas()` creates the algorithm canvases (view windows) and registers `canvasInterposeDispatch()` and `canvasEventDispatch()` to handle events within a canvas. `canvasInterposeDispatch()` catches canvas resize events and redisplays all views. `canvasEventDispatch()` catches middle and left mouse button clicks within the canvases to update `elementOfInterest` and control the animation.

`createAlgMenu()` creates the algorithm window menu and registers `menuDispatch()` with the Notifier. `algWindowHelp()` shows a help screen for the algorithm window menu and `algKill()` destroys the algorithm window.

`setAlgWindowTitle()` is called frequently by several different functions to set the algorithm window title bar to reflect the current algorithm name and recorder status.

`openFonts()` opens an array of fonts for writing text to canvases. `fonts[0]` through `fonts[5]` are empty. The remaining array elements point to fonts whose point size is equal to their array index.

int	main(int, char*)
Notify_value	frameInterposeDispatch(frame, Event, Notify_arg, Notify_event_type)
Notify_value	aaarfCommandDispatch(Notify_client, int)
void	setAlgWindowState(int)
void	createAnimationCanvas(int)
Notify_value	canvasInterposeDispatch(frame, Event, Notify_arg, Notify_event_type)
Notify_value	canvasEventDispatch(Window, Event*, caddr_t)
void	createAlgMenu()
caddr_t	menuDispatch(Menu, Menu_item)
void	setAlgWindowTitle()
void	openFonts()
void	algWindowHelp()
void	algKill()

Table 14: Function Prototypes for *aaarfCommon.c*

4.2 aaarfMaster.c

This module creates the master control panel. The master control panel is divided into four sections: control, input, algorithm, and view. The class-common library provides the control and view sections with some help from the class-specific `setBreakPointsItems()` and `setViewNames()` functions. The input and algorithm sections are provided by the class-specific `addInputSection()` and `addAlgorithmSection()` functions. The functions included in *aaarfMaster.c* are listed in Table 15.

`createMasterControl()` creates the master control panel. It calls functions to setup the four sections of the panel. It registers the `masterEventDispatch()` with the Notifier to monitor the **Help** and **Close** buttons.

`addControlSection()` sets up the control section of the master control panel. It registers `controlEventDispatch()` with the Notifier.

The `getControlParameters()` and `setControlParameters()` functions are used to save and set control parameters for the animation recorder and environment control functions. They both use the `PARAMS` structure for parameter storage. `PARAMS` is simply an array of integers. The size of the array is set by the client-programmer in the class-specific header file.

void	createMasterControl()
void	masterEventDispatch(Panel_item, struct inputevent*)
int	addControlSection(Panel, int)
void	controlEventDispatch(Panel_item, struct inputevent*)
void	getControlParameters(PARAMS)
void	setControlParameters(PARAMS)

Table 15: Function Prototypes for *aaarfMaster.c*

4.3 aaarfRecorder.c

This module implements the animation recorder and manages parameter exchanges with AAARF. Table 16 lists the functions implemented in this module and Table 17 presents the data structures defined for this module. The IE_PACKET structure provided by the client-programmer in *classSpecific.h* (see Table 20) and the SETUP_PACKET structure defined in *aaarfIPC.h* (see Table 10) are also used.

createRecorder() creates the recorder panel and registers recordDispatch(), buildMenu(), and getSelection() with the Notifier. The recorder's path and recording selectors work exactly like the path and environment selectors of the environment control panel (Section 3.4).

Recordings are stored in a linked list structure (see Figure 17). The list is managed with three pointers: head, tail, and next. playIE() provides IEs by traversing the linked list. The list is doubly linked, but the current version of AAARF does not support reverse playback. Recordings are created with saveIE() which accepts an IE, malloc()s a new node, and adds the IE. restartRecording() sets the next pointer to the head of the list. freeRecorderMemory() frees the memory previously allocated for a recording. readRecording() and writeRecording() are used for reading and writing recordings to disk.

setRecorderState() controls the recorder functions and maintains state information regarding the animation recorder.

readParameters() and restoreParameters() are used to restore the animation state. getParameters() and sendParametersToAAARF() are used to save the animation state.

void	createRecorder()
void	recordDispatch(Panel_item, struct inputevent*)
Menu	buildMenu(Menu, Menu_generate)
int	getSelection(Panel_item, Event*)
int	setRecorderState(int)
void	readParameters(SETUP_PACKET, int, int)
void	restoreParameters()
void	getParameters()
void	sendParametersToAAARF(int)
void	readRecording(int)
void	writeRecording(int)
void	restartRecording()
void	freeRecorderMemory()
IE_PACKET	*playIE()
void	saveIE(IE_PACKET*)
void	recorderHelp()

Table 16: Function Prototypes for *aaarfRecorder.c*

```
typedef struct IEnode IE_NODE;

struct IEnode
     {
            IE_NODE *lastIE;
            IE_PACKET IEpacket;
            IE_NODE *nextIE;
     }
```

Table 17: *aaarfRecorder.c* Data Structures

4.4 aaarfViews.c

This module sets up the view section of the master control panel, creates the status display panel, and manages the animation views. Table 18 lists the functions included in this module. The `VIEW_STATE` data structure is used to manage the view windows. This structure is defined in *classCommon.h* (Table 13).

`addViewSection()` creates the view section of the master control panel and registers `viewDispatch()` with the Notifier. The `getViewParameters()` and `setViewParameters()` functions are used by the recorder and the environment control to get and set view parameters. `restoreViews()` is used to restore a particular view window configuration after a restore operation.

`createStatusDisplay()` creates the status display and registers `statusDispatch()` with the Notifier. It calls `buildStatusDisplay()` to build the class-specific portion of the status display.

`main()` associates a polling timer with `animateTheAlgorithm()`, which is the heart of the animation. It gets an IE, processes the IE, and updates the displays. The polling timer is controlled by `setAlgTimer()`; it is at the end of every pass through `animateTheAlgorithm()`. The display state is controlled and managed by `setDisplayState()`. It is checked at the start of every pass through `animateTheAlgorithm()`; if the display state is not set to `ALG_GO`, nothing happens in that pass.

`resizeAllViews()` sizes all open views according to an argument from the calling function and the current view parameters.

int	addViewSection(Panel, int)
int	viewDispatch(Panel_item, unsigned int, Event*)
void	getViewParameters(PARAMS)
void	setViewParameters(PARAMS)
void	restoreViews()
void	resizeAllViews(int)
int	setDisplayState(int)
int	setAlgTimer(int)
void	createStatusDisplay()
void	statusDispatch(Panel_item, struct inputevent*)
Notify_value	animateTheAlgorithm(Notify_client, int)

Table 18: Function Prototypes for *aaarfViews.c*

4.5 aaarfUtilities.c

This module consists of functions that are common to both the AAARF main process and the algorithm animation processes. The functions are listed in Table 19.

`getScreenResolution` sets the screen height, width, and depth.

`panelFrameEventDispatch()` is a notify procedure which suppresses the default SunView menu from popping up on panels. It is associated with all panels used in AAARF.

`characterFilter()` is used to filter user input from the keyboard; it only passes numbers and alphabetic characters.

`getDirectory()` and `getSubdirectories()` generate a list of files and paths respectively.

`userWarning()` displays an alert message with the argument provided.

void	getScreenResolution(int*, int*, int*)
Notify_value	panelFrameEventDispatch(Frame, Event*, Notify_arg, Notify_event_type)
Panel_setting	characterFilter(Panel_item, Event*)
int	getDirectory(char*, char*, char*)
int	getSubdirectories(char*, char*)
void	userWarning(Frame, char*)

Table 19: Function Prototypes for *aaarfUtilities.c*

5 Class-Specific Functions

The most difficult part of developing an algorithm animation is determining how to meaningfully represent an algorithm's state. There's no procedures or rules for making such a determination; it's usually best to start with the familiar static representations found in text books. After creating a simple view, it's easier to devise more complex and possibly more interesting views. Determining the nature of the graphical representation is a matter of creativity; this section is more concerned with the mechanics of presenting the view.

The AAARF class-common library provides client-programmers a framework for developing new algorithm animations. This section describes the class-specific requirements of the framework and presents recommendations for using the framework to develop new algorithm animations. The *ArraySort* algorithm class is used as an example throughout this section. The source code for the *ArraySort* class is included in the *ArraySort* subdirectory of the AAARF distribution.

Section 5.1 describes how to set up a working directory to begin development of a new algorithm animation. Section 5.2 recommends a method for testing algorithm animations under development. Section 5.3 outlines the general requirements for a new algorithm animation. The remainder of the section describes the requirements in more detail.

- *The program development methods presented in this section are just suggestions. Experienced programmers may prefer variations of these methods or even a completely different approach to program development. However, adherence to the suggested methods insures some degree of uniformity among algorithm class implementations, making it easier for any programmer, regardless of their experience level, to modify or complete any another programmers project.*

5.1 Creating a Working Directory

Begin by creating a working directory. Copy the contents of the *ArraySort* directory to use as a template for the development. Either edit the *ArraySort* files or create new files using the *ArraySort* files as a guide. In either case, existing algorithm classes are probably the best guide for developing new algorithm classes. Use the UNIX *ln* command to create a symbolic link to the AAARF class-common library. Figure 9 shows typical commands for setting up a working directory; don't take this example too literally; the local AAARF directory may be in a different path.

```
DALI<1> mkdir BinPack
DALI<2> cd BinPack
DALI<3> cp /usr/local/src/aaarf/ArraySort/* .
DALI<4> ln -s /usr/local/src/aaarf/lib/*.c .
DALI<5> ln -s /usr/local/src/aaarf/lib/*.h .
```

Figure 9: Creating a Working Directory for the Bin Packing Class

5.2 Testing a New Algorithm Class

Test new animations by creating a local AAARF class file that includes the name and path of the algorithm class under development. Copy the default AAARF class file to a local file, add the new algorithm class to the local file's list of classes, and set the AAARFCLASSES environment variable to the local AAARF class file. Now when AAARF is evoked, it uses the local AAARF class file. This allows programmers to work on animations without interfering with other AAARF users. When the animation is ready for public distribution, add its name and path to the default AAARF classes file.

5.3 General Requirements

There are three categories of requirements for developing an algorithm class:

- The class-common framework requires 19 class-specific functions, 5 constant definitions, and 1 data structure definition. The functions can be distributed among any number of arbitrarily named modules, but the constants and the data structure must be in a file called *classSpecific.h* (see Section 5.4).

- There must be a `paintXXX()` and an `updateXXX()` function for every view that is implemented.

- There must be a background process that implements the algorithms, generates input for the algorithms, and maintains a communications link to the window-based process for reporting IEs generated by the algorithms.

  In general, the background process consists of a `main()` function to establish communications with the window-based process and to call the input generator and the appropriate algorithm, an IE Dispatcher to send interesting events to the window-based level, and the algorithm functions. Figure 10 shows the general structure of the background process.

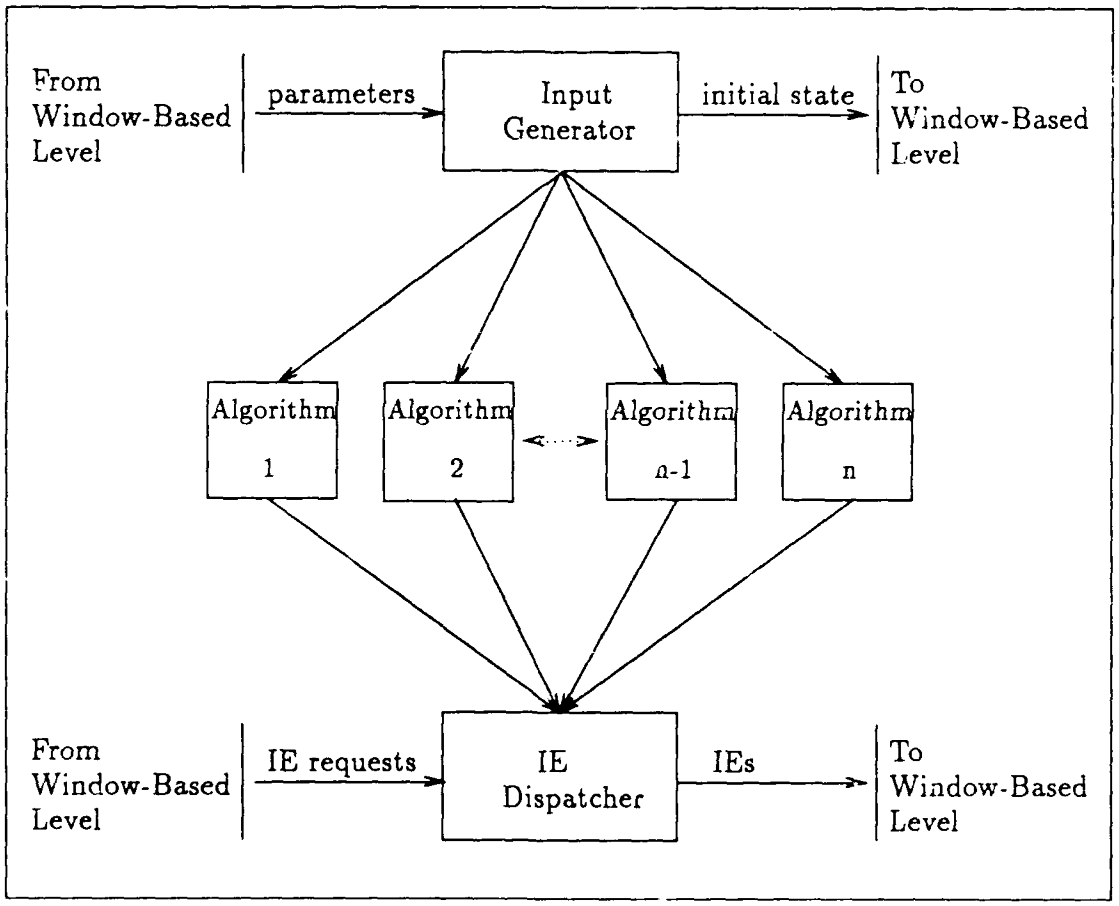


Figure 10: Background Process Structure

5.4 Class-Specific Header File

The AAARF class-common functions need five class-specific constants and one class-specific data structure. The data structure is used to report IEs from the background process and to record and playback IEs on the animation recorder. There is no limitation to the size or contents of the structure, but it must be of the type, `IE_PACKET`. Table 20 shows the *classSpecific.h* header file for the *ArraySort* algorithm class.

The five constants are:

- `CLASS_NAME` The name of the algorithm class.
- `CLASS_ICON` The file name of the class icon. *iconedit*, an icon editor available on most Sun workstations, can be used to create an icon. Alternatively, any icon in the *icons* subdirectory can be use.
- `TIMER_SCALE` Sets a multiplier for the number of milliseconds between calls to `animateTheAlgorithm()`. It may take a few iterations to get this number just right; 3000 is a good starting point. The number must be greater than zero.
- `TIMER_OFFSET` Sets the minimum number of milliseconds between calls to `animateTheAlgorithm()`. This number must also be greater than zero; one is a good starting point.
- `PARAM_SIZE` Sets the number of elements in the parameter storage array.

```
#define CLASS_NAME                    "ArraySort"
#define CLASS_ICON                    "ArraySort.icon"

#define TIMER_SCALE                   3000
#define TIMER_OFFSET                  1

#define DEFAULT_VIEW                  1
#define PARAM_SIZE                    50

typedef struct
    {
        int type;
        int index1;
        int index2;
    }
    IE_PACKET;
```

Table 20: Sample *classSpecific.h* from the ArraySort Class

5.5 Input Functions

Three input functions are required by the class-Common library; they are listed in Table 21. addInputSection() uses SunView panel items to provide a mechanism for setting parameters to the input generator. The master control panel handle and a row number within the panel are passed to the function. Panel items are positioned beginning at the row number. After all panel items are placed, the row number at which the next panel section can begin is returned. Notify procedures for the input section are not required, but may be used if deemed necessary by the client-programmer.

getInputParameters() and setInputParameters() provide a means for saving and setting parameters to support the recorder and the environment controller.

int	addInputSection(Panel, int)
void	getInputParameters(PARAMS)
void	setInputParameters(PARAMS)

Table 21: Required Input Functions

5.6 Algorithm Functions

Four algorithm functions are required for the class-common library; they are listed in Table 22. One function is for adding algorithm panel items to the master control panel, and two functions are used to save and restore the algorithm parameters. The fourth function, **getAlgorithmName** is used to post the algorithm name on the algorithm frame title bar. It returns a pointer to the currently selected algorithm name. A notify procedure for the algorithm panel is not required, but can be used if deemed necessary by the client-programmer.

int	addAlgorithmSection(Panel, int)
char	*getAlgorithmName()
void	getAlgorithmParameters(PARAMS)
void	setAlgorithmParameters(PARAMS)

Table 22: Required Algorithm Functions

5.7 View Functions

The class-common library only requires three view functions, but for every view supported by an algorithm class, there must also be a function to paint and update the view within an arbitrarily sized view window. Table 23 lists the required view functions. When one view requires painting or updating, they all do; thus the two functions `paintAllViews()` and `updateAllViews()`. Since the update operation depends on the current IE, `updateAllViews()` requires an IE as an input argument.

`processIE()` accepts the current break point setting and the current IE and returns the animation state: stop, continue, or finished. Within the routine, the animation data structure is modified in accordance with the current IE.

int	processIE(int, IE_PACKET*)
void	paintAllViews()
void	updateAllViews(IE_PACKET*)

Table 23: Required View Functions

5.8 Status Functions

Four status functions are required; they are listed in Table 24. The status functions share several global variables with the view functions, so they are typically grouped together in the same module. *ArraySortView.c* is an examples of a module that combines the view and status functions.

`buildStatusDisplay()` creates the class-specific entries in the status display. The status display handle and a row number are input parameters. `statusHelp()` provides a help screen for the status display.

`updateElementOfInterest()` is closely related to the view functions in that it must determine which element or aspect of an animation a user has selected.

`updateStatus()` is also related to the view functions - `processIE()` updates the appropriate variables for the status display. The updates are applied to the status panel when this function is called.

void	buildStatusDisplay(Panel, int)
void	updateElementOfInterest(int, int, Window)
void	updateStatus()
void	statusHelp(Frame)

Table 24: Required Status Functions

5.9 Control Functions

There are five control functions required by the class-common library. They are listed in Table 25. The `setViewItem()` and `setBreakPointItem()` functions are used to set the class-specific portions of the view section and control section of the master control panel. `masterHelp()` provides the class-specific help screen for the master control panel.

`runAlgorithmInBackground()` is an important function that manages the algorithm state structure and the class-specific background process.

`getIE()` is the communications link to the background process. It sends an IErequest and receives the next IE.

void	runAlgorithmInBackground()
IE_PACKET	*getIE()
void	setBreakPointItem(Panel_item)
void	setViewItem(Panel_item)
void	masterHelp()

Table 25: Required Control Functions

6 AAARF Projects

This section presents suggestions for AAARF extensions and animation projects.

1. Extend the AAARF class file to include other configuration variables that are currently either compile time defines, hard-coded defines, or environment variables. For example, maximum number of windows; maximum number of views; maximum and minimum window size; default recording and environment paths; default algorithm window color codes; etc. Most likely, two configuration files are required: one for the AAARF administrator and one for the user.

2. Implement a "snap" option which forces the width, height, and position of windows to be a multiple of some number of pixels. This feature makes it easier to create windows of the same size and to align them neatly for comparisons.

3. Modify the animation recorder such that reverse playback is supported. This allows the end-user to immediately review an interesting part of a (recorded) animation, without restarting the recording.

4. Extend the animation recorder's capabilities such that every animation is always recorded. This allows the user to use the reverse button at any time during the animation. An interesting part of an animation could be reviewed immediately (currently, the user must wait for the algorithm to finish before playing the recording). When the user hits the reverse button, the recorder provides IEs to the view functions; the background procedure continues to wait with its next IE. If the animation returns to the point at which the reverse button was hit, IEs are again retrieved from the background procedure.

5. Extend the capability of the recorder such that it records control events. For example, detect and record speed changes, changes in the break point setting, user requested stops, etc. This permits end-users to create recordings that emphasize a particular event or set of events within an animation.

6. Develop a linkable library of animation objects similar to that developed by John Stasko in TANGO [6]. The library should provide a set of primitive graphic objects and functions to support smooth animation of the objects between two points or along a predetermined path. This could greatly simplify the development of the view functions, arguably the most grueling programming task for the client-programmer.

7. Modify the AAARF class-common library such that it is a linkable library.

8. Add a *current state* indicator to the algorithm window structure. The indicator reflects the current state of the algorithm (running, stopped, finished, etc) and the current state of the recorder (recording, reverse, forward, paused, etc). Unlike the status display panel which may be opened or closed, this indicator is always displayed. The information can appear on the title bar or within some reserved area in the algorithm window.

9. Modify AAARF's color table usage so that pixrect patterns are used in addition to color. This should provide more interesting displays on monochrome monitors.

10. Develop a bin sorting algorithm class. Algorithms should include first-fit, best-fit, worst-fit.

11. Develop a tree searching program that animates various type of search algorithms: depth-first, breadth-first, etc.

12. Animate several numerical approximation methods such as finding roots, integration, least common denominator.

13. Animate the "Drinking Philosophers" problem using a petri net representation.

References

[1] Brown, Marc H. *Algorithm Animation.* Cambridge, Massachusetts: The MIT Press, 1987.

[2] Fife, Keith C. "The AAARF User's Manual." Air Force Institute of Technology, November 1989.

[3] Fife, Keith C. *Graphical Representation of Algorithmic Processes.* MS thesis, Air Force Institute of Technology, 1989.

[4] Hartrum, Thomas C. *System Development Documentation Guidelines and Standards.* Draft #4, Air Force Institute of Technology, January 1989.

[5] Kernighan, Brian and Dennis Ritchie. *The C Programming Language.* Prentice-Hall, Inc, 1978.

[6] Stasko, John T. "Simplifying Algorithm Animation Design with TANGO." Georgia Institute of Technology, June 1989.

[7] Sun Microsystems. *C Programmer's Guide*, 1988. SunOS Technical Documentation.

[8] Sun Microsystems. *Getting Started with SunOS: Beginner's Guide*, 1988. SunOS Technical Documentation.

[9] Sun Microsystems. *Network Programming*, 1988. SunOS Technical Documentation.

[10] Sun Microsystems. *Pixrect Reference Guide*, 1988. SunOS Technical Documentation.

[11] Sun Microsystems. *Setting Up Your SunOS Environment: Beginner's Guide*, 1988. SunOS Technical Documentation.

[12] Sun Microsystems. *SunView1 Beginner's Guide*, 1988. SunOS Technical Documentation.

[13] Sun Microsystems. *SunView1 Programmer's Guide*, 1988. SunOS Technical Documentation.

Appendix E. *Source Code*

The AAARF source code is included in Volume 2.

Vita

Captain Keith C. Fife [redacted] attended the University of Kentucky for two semesters. He enlisted in the United States Air Force in February 1974 and attended the Defense Language Institute, where he graduated with honors from the Arabic-Egyptian language course. After working as a Voice Processing Specialist for two years at Iraklion Air Station, Crete, Greece, he returned to the Defense Language Institute to study North Vietnamese. He graduated with the highest honors, and went on to teach National Intelligence as an Air Training Command instructor at Goodfellow Air Force Base. He was accepted into the Airman Education and Commissioning Program in June 1982 and enrolled at the Ohio State University. He graduated Magna Cum Laude with the degree of Bachelor of Science in Electrical Engineering in April 1985. After graduation, he worked as an Electronic Warfare Engineer at McClellan Air Force Base, California. He entered the Air Force Institute of Technology in June 1988.

[redacted]

UNCLASSIFIED

Traditionally, software engineers have described algorithms and data structures using graphical representations such as flow charts, structure charts, and block diagrams. These representations can give a static general overview of an algorithm, but they fail to fully illustrate an algorithm's dynamic behavior. Researchers have begun to develop systems to visualize, or animate, algorithms in execution. The form of the visualization depends on the algorithm being examined, the data structures being used, and the ingenuity of the programmer implementing the animation.

This study chronicles the development of the AFIT Algorithm Animation Research Facility (AAARF). The goal of the study is (1) to develop a method for developing algorithm animations and (2) to develop an environment in which the animations can be displayed. The study emphasizes the application of modern software engineering techniques. The development follows a classic life-cycle software engineering paradigm: requirements, design, implementation, and testing. An object-oriented approach is used for the preliminary design. The system is implemented with the C programming language on a Sun workstation and uses the SunView window-based environment.

A framework is proposed for developing algorithm animations that minimizes the development time for client-programmers. The framework also ensures that end-users are presented a consistent method for interacting with the animations. The algorithm animation environment provides end-users with a variety of control and viewing options: multiple algorithms, multiple views, simultaneous control of algorithms, and animation environment control. Three levels of execution are used to provide multiple algorithm animations and multiple views of algorithms. The animation manager and view generators must reside on the Sun workstation, but the algorithms can reside on any network-connected host.

UNCLASSIFIED

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): AFIT/GCE/ENG/89D-2	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: School of Engineering — 6b. OFFICE SYMBOL (If applicable): AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code): Air Force Institute of Technology, Wright-Patterson AFB, Ohio 45433-6583	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Phase 1 Program Office — 8b. OFFICE SYMBOL (If applicable): SDIO	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code): Wright-Patterson AFB, Ohio 45433	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
THE GRAPHICAL REPRESENTATION OF ALGORITHMIC PROCESSES (UNCLASSIFIED)

12. PERSONAL AUTHOR(S)
Keith C. Fife, Capt, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
MS Thesis	FROM ____ TO ____	1989 December	216

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP
12	01	
12	05	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Computer Programs, Computer Graphics, Algorithms, Computer Aided Instruction

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Thesis Advisor : Gary B. Lamont
Professor
Department of Electrical and Computer Engineering

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Dr. Gary B. Lamont, Professor	22b. TELEPHONE (Include Area Code): (513) 255-2024 — 22c. OFFICE SYMBOL: AFIT/ENG

DD Form 1473, JUN 86 Previous editions are obsolete. SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED